Programming Interactive Real-Time Games over WLAN for Pocket PCs with J2ME and .NET CF

Andreas Janecek
Institute of Distributed and Multimedia Systems
University of Vienna,
Lenaugasse. 2/8,
1080 Vienna, Austria
Tel.: +43 1 4277 39650, Fax: +43 1 4277 39651
andreas.janecek@univie.ac.at

Helmut Hlavacs
Institute of Distributed and Multimedia Systems
University of Vienna,
Lenaugasse 2/8,
1080 Vienna, Austria
Tel.: +43 1 4277 39659, Fax: +43 1 4277 39651
helmut.hlavacs@univie.ac.at

ABSTRACT

In this paper we compare the Java 2 Micro Edition and the .NET Compact Framework with respect to their performance for programming multiplayer games for PDAs. We benchmark results for both platforms dealing with computational capabilities and communication performance over WLAN. We also have developed a simple prototypical multiplayer game called 3D-Pong and evaluate how simple resilience mechanisms can hide WLAN packet losses for this game.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**] Performance Analysis and Design Aids

D.3.3 [**Programming Languages**]: Language Constructs and Features – *constraints, frameworks, input/output.*

General Terms

Measurement, Documentation, Performance, Design, Reliability

Keywords

Interactive Real-Time Games, Java Micro Edition, .NET Compact Framework, WLAN, PocketPC,

1. INTRODUCTION

One of the dominating operating systems for personal digital assistants (PDAs) is given by Microsoft's Pocket PC (PPC), now called Windows Mobile 2002/2003, all being based on Windows CE. Due to the performance increase of Pocket PCs in the last years, more and more games have been developed for this platform, usually for one player only [7]. However, PDAs are gradually being equipped with wireless communication, for instance with Bluetooth/WLAN, either built in or via cards for compact flash or (if available) SDIO slots. Alternatively, a Pocket PC Phone Edition incorporates GPRS or UMTS. This wireless connectivity now enables interactive gaming with other users in (almost) real-time. Some vendors have produced commercial interactive real-time games or gaming middleware [9, 4].

However, GPRS and UMTS still show latencies which are too high for enabling action-type real-time games like car races, sports games or shooters, which depend on tight upper limits for latency [2], and only enable interactive games with softer latency limits like strategy games [11]. A possible solution for such a dilemma might be the use of a hybrid approach, where game hosting and billing is carried out via UMTS, while real-time communication between the mobile stations is done via WLAN [5]. WLAN (mainly IEEE 802.11b) right now seems to be a good candidate for real-time communication for mobile games, due to its wide availability, low price, high performance, low latency and reliability. It can be used in infrastructure mode with fixed access points, or multiple mobile nodes may construct an ad-hoc network with dynamic routing over multiple hops. The latter approach introduces additional delay, bandwidth decrease and unreliability, which may negatively influence the interactivity for real-time games [6].

An important question for game designers of course also is given by the choice of the software platform they use. In the past, besides native programming, two popular choices have been developed for Pocket PCs, the Java 2 Micro Edition (J2ME) from Sun and the .NET Compact Framework (.NET CF) from Microsoft. The main advantage of Java is given by its portability, although due to the variability of mobile platforms, the claim "write once, run everywhere" cannot be achieved by 100%. .NET CF is fixed for Pocket PC, but offers a better integration into Microsoft's operating system Windows and the new .NET paradigm.

For our investigation we were interested into the question which software platform (J2ME or .NET CF) delivers the better performance for certain computational tasks we implemented for both platforms. Hence, for graphical output we did not use Microsoft's Game API (GAPI) [17], but used routines available for each platform instead. Furthermore, we used the IP measurement tool CODIS Net [3] to measure the raw communication performance of our PDAs. Measurements were carried out on an older Pocket PC using Windows Mobile 2002 and a more modern Pocket PC running Windows Mobile 2003. Other benchmarks for Pocket PC in general and GAPI in particular can be found, for instance, at [1].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
NetGames'05, October 10–11, 2005, Hawthorne, New York, USA.
Copyright 2005 ACM 1-59593-157-0/05/0010...$5.00.

As a case study, we developed an interactive real-time game called *3D-Pong* for each of the software platforms. Here, we further evaluated the game performance and the general bandwidth requirements for such a game, and also evaluated the use of simple strategies for coping with the inevitable loss of packets occurring in WLANs.

2. JAVA 2 MICRO EDITION

The Java 2 Micro Edition (J2ME) defines a set of application programming interfaces (APIs) for the Java language, especially designed to run on mobile computers like PDAs and mobile phones, or on settop boxes, embedded systems, etc. Just like the Standard Edition API is a subset of the Enterprise Edition API, the Micro Edition API is mainly a subset of the Standard Edition API, with the exception of a few special packets which are integrated into J2ME only, like for instance for the communication via the infrared interface. One big problem is given by the variability of the hardware resources found in mobile computers with limited capability, like memory size, CPU power, system design, availability of a keyboard etc. In order to cope with this variability, the J2ME consists of three layers (*virtual machine*, *configuration* and *profile*) which can be customized to the specific hardware needs of the used mobile platform.

The Java Virtual Machine (JVM) for J2ME is placed at the lowest layer and exists in two versions. The CDC (Connected Device Configuration) HotSpot Implementation is a complete virtual machine comparable to the one of J2SE and is designed for mobile devices with medium capabilities like high-end PDAs with a memory size of several MBs, large display and modern ARM CPU [13]. On the other hand, the CLDC (Connected Limited Device Configuration) HotSpot Implementation is designed to run on mobile equipment with tight hardware restrictions, for instance a memory size of only a few hundred KB [14]. The CLDC JVM meets the JVM specifications, but requires only 40-80 KB of memory. Thus, it can be used, for example, in low-cost PDAs, pagers or current mobile phones.

The second layer of J2ME defines *configurations* for different types of mobile hardware resources, thus yielding a *horizontal* classification. As a result, platforms being as different as for instance settop boxes and mobile phones may be described by a single configuration, if the same type of hardware is used. A configuration is therefore a combination of a JVM and a set of APIs which are useful for the described type of hardware. The currently only available configurations are given by CDC and CLDC. For example, since CLDC defines APIs for low-cost PDAs or mobile phones with little memory, it does not include features like advanced error handling, user-defined class loaders or floating-point support.

In contrast to the second level which defines different hardware types, the third level uses a *vertical* classification scheme called *profiles* for the distinction of different types of functionality, like mobile phone, PDA, washing machine etc. Profiles define APIs specifically designed for the profile use cases. For instance, the Personal Basis Profile (JSR 129) and additionally the Personal Profile (JSR 62) define AWT support, thus enabling GUIs for interactive applications. For CLDC, currently only the Mobile Information Device Profile (MIDP, JSR 37) exists, which for example supports a simple (non-AWT) user interface called LCDUI.

3. .NET COMPACT FRAMEWORK

Similar to J2ME, the .NET CF is a subset of the common .NET framework, the necessary amount of memory has been decreased from 21 MB for .NET to only 2 MB for .NET CF [16]. It currently supports the platforms Pocket PC, Pocket PC 2002 and all versions of Pocket PC 2003 (aka Windows Mobile 2003) including Phone and Smartphone Edition. Programs may be developed using Visual Basic.NET or C#.NET. In contrast to the normal .NET framework, however, both languages may not be used in parallel for one program. As usual for .NET, applications are first compiled into a platform independent bytecode called Common Intermediate Language (CIL), which then is moved to the Pocket PC. In order to run the bytecode, the Common Language Runtime (CLR) must be installed, which is similar to the Java Runtime Environment. On the Pocket PC, the CLR uses a JIT to again compile the bytecode into native machine language, which is then executed. CLR-based code is also called *managed code*, since the CLR is able to enforce security restrictions for program hardware access. Also, the CLR manages memory allocations, acts as garbage collector and protects array boundaries. Code which is able to directly access the hardware is called *unmanaged* and can only be written in C++.

CF managed code can call unmanaged code (implemented in a DLL) by using the Platform Invoke (P/Invoke) mechanism. P/Invoke calls may only result in an integer value returned from the DLL, which for instance is the case for most GAPI methods. However, some GAPI methods return data structures which calls for the use of a C++ wrapper DLL.

There are many restrictions for the .NET CF with respect to the standard .NET framework: method overloading is reduced, many controls, for instance for printing, have been omitted, XML and database support is reduced, no windows registry support, no role-based security model, no access to COM or ActiveX components, etc.

4. COMPARISON OF THE TWO FRAMEWORKS

The J2ME and the .NET Compact Framework are the two leading platforms for application-development on smart devices. Both platforms offer some related benefits to improve developer productivity but there are many refined distinctions.

All programming languages are well known (Java vs. C#.NET and VB.NET), and like their big counterparts on the desktop both technologies run their programs inside managed runtime environments that manage memory usage, security and runtime optimization, and use rich libraries and components for the reuse of software modules. The APIs used in J2ME/CDC and .NET CF are a subset of the J2SE resp. the .NET framework, and offer additionally optional packages for developing programs on smart devices. .NET CF programs use a simplified .NET security model while J2ME/CDC supports the full Java security manager. CLDC programs only use a limited security model.

The used IDEs are well known and extended with tools for programming applications for smart devices. For J2ME a variety of commercial and non-commercial IDEs are available (e.g., IBM WebSphere Studio Device Developer, J-Builder, Eclipse, vendors SDKs, etc.), for .NET CF development the Visual Studio.NET is mandatory, thus requiring an expensive IDE for writing programs for Pocket PCs under the .NET CF. On the other side, the

VS.NET 2003 is well designed and manages the installation of the runtimes and class libraries, has an emulator in which programs can be tested and provides good support for remote debugging.

Beside that, the supported hardware is the main difference between both platforms. While J2ME programs are able to run on various devices, .NET programs are only supported by Pocket PCs running a Windows OS. Both platforms cover a limited set of important and frequently used features, device-specific features are accessible via native methods. J2ME/CDC applications can access native methods through the Java Native Interface (JNI) [12] while CLDC does not support native methods. .NET CF programs invoke methods in specifically formatted Win32 native libraries using the P/Invoke command. Because Microsoft is responsible for the operating system and the .NET CF as well, .NET CF has a better support for native methods. Examples for the use of P/Invoke are Email and PIM support (using the Outlook API), SMS, Instant Messenger or Windows Media APIs. J2ME/CDC uses Java Phone, third party APIs and the Wireless Messaging API for Email, PIM and SMS. XML and Web Services are supported directly in the .NET CF while Java needs third party tools to use these technologies. Web Services tools are integrated with Visual Studio.NET, kSOAP plug-ins provide Web Service tools for Java development.

5. BENCHMARK RESULTS

In our benchmarks we measure the difference in performance between J2ME and .NET CF with respect to the hardware resources and operating systems of the used Pocket PCs. We use two Pocket PCs that differ in hardware resources like memory and processor speed, and also the operating systems versions. The Fujitsu Siemens Pocket Loox 600 running Pocket PC 2002 has only 64 MB of memory and uses a PXA250 Intel processor with 400 MHz. Our second PDA, a Toshiba PocketPC e800PT with 126 MB of memory uses an Intel PXA263 processor with also 400 MHz running Windows Mobile 2003 (Pocket PC 2003).

For the Java programs we used the J9 VM as our runtime environment, which is IBM's implementation of the Java Virtual Machine Specification and the core of IBM's WebSphere Everyplace Micro Environment (WEME) [21]. The WEME product is supported on a variety of operating systems, (e.g., Microsoft Windows, Linux, PalmOS, Pocket PC, Symbian) and various hardware architectures (e.g., Intel x86, xScale, ARM, MIPS). On the Microsoft-side we used the .NET Compact Framework 1.0, which is included into the Visual Studio .NET 2003 or can be downloaded from [15].

In addition to running standard CLASS and JAR files, the J9 VM can also run J9 executable (JXE) files, a format that is optimized to execute on the J9 more efficiently than standard classfiles. JXE files may be stored and executed "in place" in ROM, minimizing application startup times and RAM requirements. We compiled the Java programs for both JAR and JXE, to determine performance differences between these two formats. The .NET programs are written in C#.NET and only consist of managed code, thus no native functions are used.

In our benchmarks we measure the computing and the communication performance, which are both significant parameters for interactive real-time games. The measurement of the computing performance was split into numerical performance and graphical display performance. In our case study an adequate numerical performance is needed to compute the new positions of the ball and the rackets while the performance for graphical display has to comply with the requirements needed for real-time games, like high frame-rates.

5.1 Computing Performance

Our first benchmark suite measures the integer performance and different methods for graphical representation for strings and low-level drawing elements. Figure 1 to Figure 5 show the mean of five runs for the JAR, JXE and C#.NET programs, whereas big outlier results have been cut out.

Benchmark one multiplies two $N \times N$ integer matrices with each other, which needs a total amount of $2N^3$ integer operations to compute. As can be seen in Figure 1, the results for this test are closely related for all file formats on both platforms. Only the JAR version on the Fujitsu Siemens Pocket Loox 600 (FS) is obviously faster than its opponents (for the difficult case of $N = 200$).

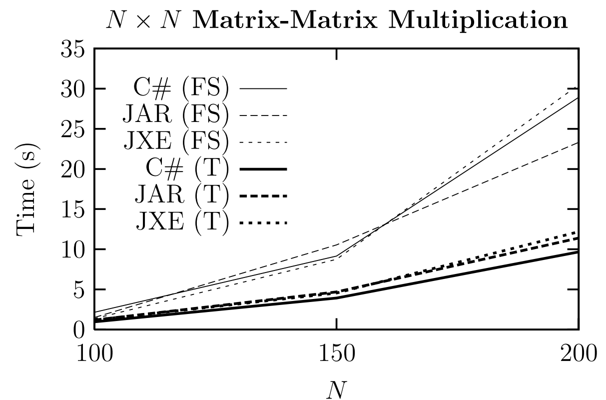

Figure 1: Benchmark 1: $N \times N$ matrix-matrix multiplication.

For the computation of *N* transcendental functions (Figure 2) the C# program on the Fujitsu Siemens PDA needed nearly twice the time of the JXE program. On the more powerful Toshiba PDA (T) the values did not differ much.

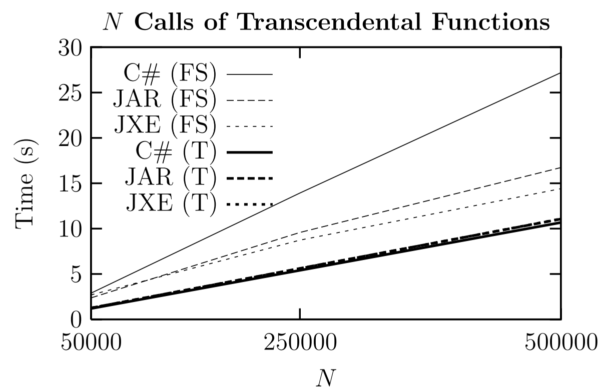

Figure 2: Benchmark 2: *N* calls of transcendental functions.

As we changed the subject of our benchmarks to graphical representation (Figure 3 to Figure 5) the results on the Fujitsu Siemens PDA turned around. C# now can achieve much better results compared to the Java programs. For both the graphical representation of a string and a low-level element like an ellipse, the average duration of C# was about half the duration of the JAR

and JXE files. On the Toshiba device the results remain constant, C# has always the best performance but the Java programs are close behind. Only for the simulation of a ball (as used in the 3D-Pong game), C# can achieve a much better performance than its opponents (Figure 5).

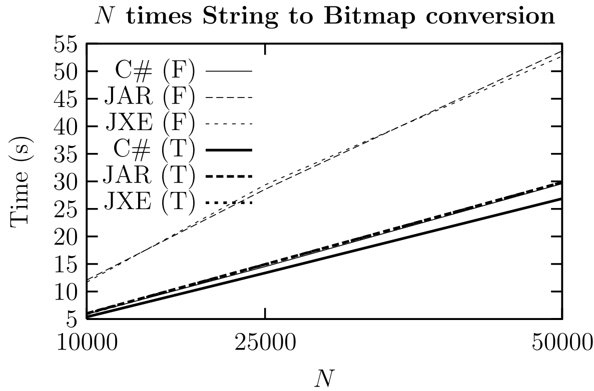

Figure 3: Benchmark 3: *N* string to bitmap conversions.

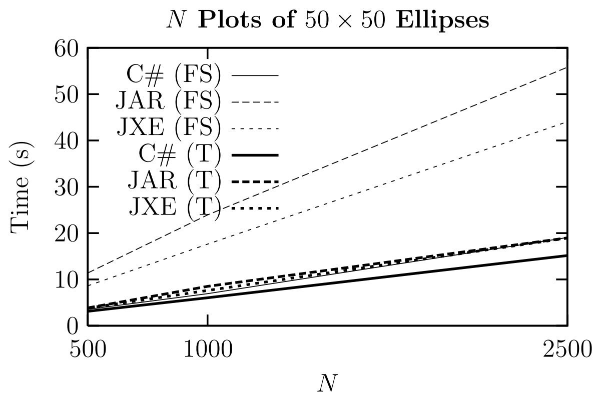

Figure 4: Benchmark 4: Drawing *N* ellipses of size 50×50 pixels.

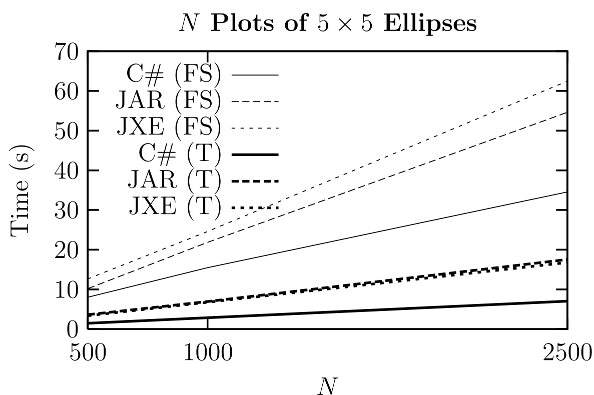

Figure 5: Benchmark 5: Drawing *N* ellipses of size 5×5 pixels.

Looking at the results from the two PDAs we can identify three differences: First, except for the durations of the C# program of benchmarks 3 and 4, all programs can gain a massive improvement in their performance on the more powerful PDA. Second, in contradiction to the runtimes on the slower PDA, C# has the best performance in all benchmarks on the Toshiba PDA, although there is no big gap to Java. On the Pocket Loox 600 Java can obtain better runtimes for the numerical performance benchmarks, but falls far behind C# for the benchmarks that deal with graphical display. Third, the differences of the standard variances of the measured durations on the two devices are enormous. Of course, as the durations get smaller the variances are expected to be lower, but the variances for the JAR program for the benchmarks 4 and 5 (graphical representation) are about 50 times larger on the slow PDA.

The results can be summarized as follows: On PDAs with relatively weak resources like the Fujitsu Siemens Pocket Loox 600, C# has much more potential with respect to graphics display, although Java has better numerical performance. On theses devices C# turns out to be the better alternative for writing real-time games than Java, because a game's success depends on its painting routines. On powerful devices with more computing power and memory, Java nearly reaches the performance of C# in most cases. The simulation of a flying ball (5×5 pixels) is the only benchmark, where C# has an obviously better performance. On these devices both approaches are useful alternatives.

5.2 Communication Performance

Our second benchmark suite tests the communication performance of the investigated PDAs. As measurement tool we used the Java based CODIS Net [3], which consists of a server and a client part. The server (running on the PDA) sends data to the client, which presents statistics of the received data, like received KBit/s, packets/s, lost packets/s, jitter etc., in its graphical user interface. For both PDAs we used Fujitsu Siemens WLAN (IEEE 802.11b) cards for their compact flash slots.

In our experiments we tested the raw TCP and UDP sending performance, which, as turned out, for UDP depends on the used packetrate. For the TCP performance, the server simply opens a TCP connection to the client and sends data of a certain size through the TCP connection. The client observes the amount of data coming out of the TCP pipe for each second, together with an average. The average is necessary since the TCP bandwidth is managed by the TCP congestion control and wobbles around the average bandwidth. The Fujitsu Siemens reached around 2545 Kbit/s, while the Toshiba reached around 2830 Kbit/s average TCP goodput.

For measuring the UDP throughput, CODIS Net allows to specify the requested bandwidth and the used number of UDP packets per second. Since for a fixed packetrate and growing bandwidth the UDP packet sizes grow larger and larger, the user can also specify a maximum size for the UDP packets (MTU). If the UDP packets exceed the MTU, they are split into smaller packets, resulting in a much higher gross packetrate.

The results for the Pocket Loox 600 with no MTU specified are shown in Figure 6, while Figure 7 shows the observed UDP bandwidth for MTU=1400 bytes. This value of 1400 bytes is chosen such that each UDP packet fits into a WLAN frame. As for a limited MTU more packets are sent because packets are split up if they exceed the MTU, it can be seen that for the slow PDA, the UDP performance heavily depends on the used packetrate.

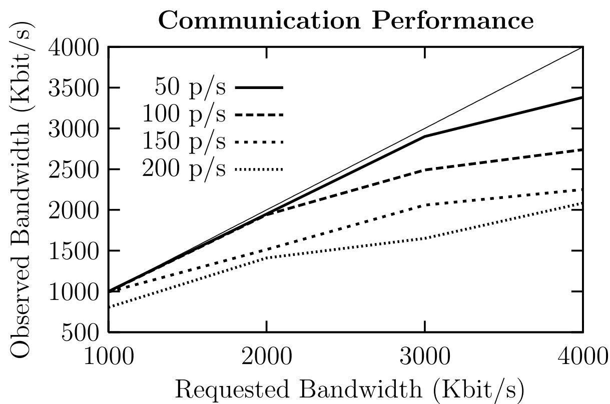

Figure 6: Benchmark 6: UDP performance of Fujitsu Siemens with no MTU specified.

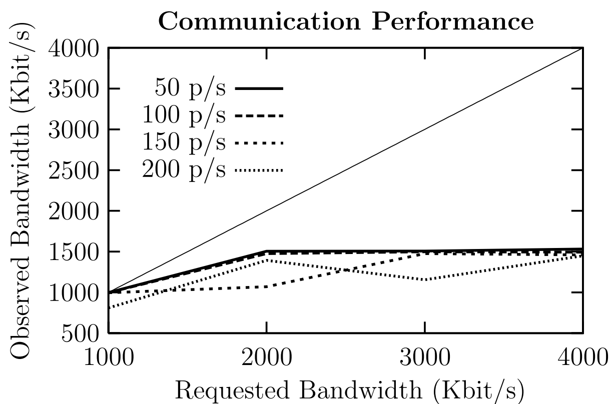

Figure 7: Benchmark 7: UDP performance of Fujitsu Siemens with MTU=1400 bytes.

Here, the limiting computing capabilities severely limit the possible UDP bandwidth. The observed bandwidth levels off at about 1500 KBit/s for 50 packets/s, if more packets are sent, the bandwidth further decreases. Hence, the adaptive strategy for coping with packets losses from Section 6.2 can by a useful strategy for low-level devices.

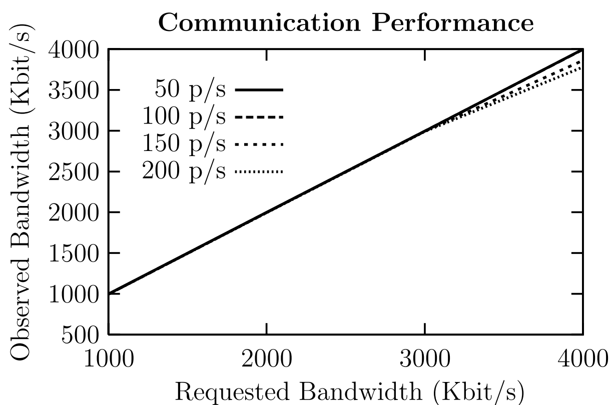

Figure 8: Benchmark 8: UDP performance of Toshiba with no MTU specified.

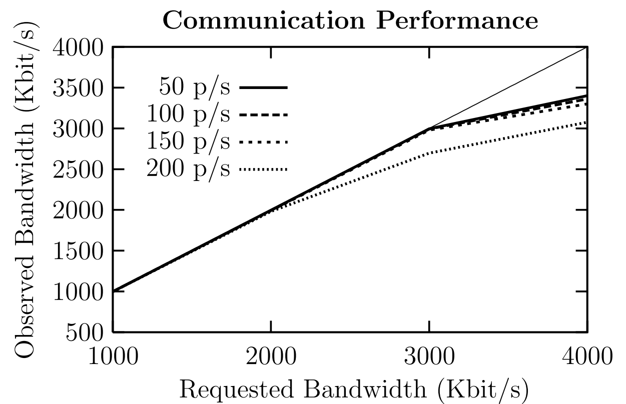

Figure 9: Benchmark 9: UDP performance of Toshiba with MTU=1400 bytes

The respective results for the Toshiba PDA are shown in Figure 8 and Figure 9. The more powerful Toshiba does not depend on the packetrate to the same extent as the Fujitsu, but a growing restriction due to a specified MTU is observable (Figure 9).

As our 3D-Pong game has rather low packet rates (about 20 packets/second at the most) and the information in each packet is not very big it can easily fit the restriction for the communication performance of the used PocketPCs.

6. CASE STUDY: 3D-PONG

Our case study focuses on a game called 3D-Pong, a dual-player real-time table-tennis game extended to three dimensions (Figure 10).

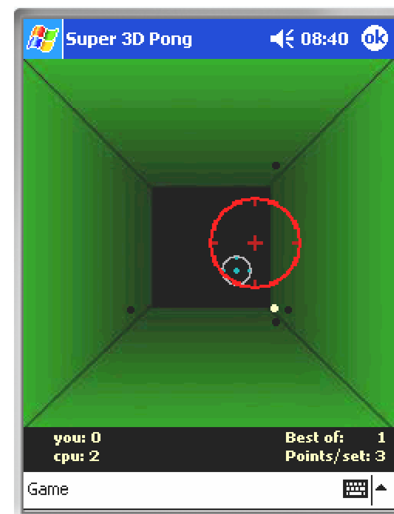

Figure 10: 3D-Pong game window.

The game was not directly ported to be played on Pocket PCs, but rather was newly developed and optimized for the use on resource-limited smart devices. 3D-Pong is implemented in J2ME/Personal Profile for Java and C#.NET for the .NET Compact Framework. It does not use any additional APIs like the GAPI for .NET CF or optional packages for the J2ME, to maintain platform independence and not to blur the results between the basic features of J2ME and .NET CF.

A peer-to-peer-like architecture is used for the game, but there are two kinds of players, a master and a slave, which differ in their capabilities. While the slave only sends his racket-positions in a fixed interval to the master, the master is responsible for the game. He calculates the position of the ball and checks if a ball has left the game area or has hit a racket or a wall. The master sends different information to the slave, including an ID and the state of game. The rest of the sent information depends on the state of game. For the game we use an IEEE 802.11b WLAN-connection, which was set up in infrastructure-mode to be able to measure the link quality and the signal level, but playing in ad-hoc-mode is possible as well.

6.1 Real-Time Games: J2ME vs .NET CF

During the development of the game we gained much experience referring to programming interactive real-time games with J2ME and .NET CF. Real-time games require high frame-rates to appear fluently. Providing all that graphical activity at high frame-rates while responding quickly to user inputs are the main challenges when programming such kinds of games. For interactive games the communication with other programs is also an important factor. The programming languages have to meet these requirements to provide a useful framework for the development of interactive real-time games. The following results refer to software development on Windows-based Pocket PCs only, since .NET CF programs cannot run on other platforms.

As shown in Section 5, C# has a better graphic-routine-performance than Java. The gap between C# and Java on slow PDAs is enormous. On powerful Pocket PCs Java can obtain nearly the same results than C#. Double-buffer graphic-routines could be used in both approaches, drawing transparent images is somewhat laborious in C#, because raster images like bitmaps are represented in rectangular format and have to be translated to irregular shapes using the *Color Key* technique. Event-handling and fast reactions on user inputs are similar to particular desktop-applications in both languages. For the WLAN-communication the well known desktop-program methods could be used.

The advantages of the .NET Compact Framework for the development of interactive real-time games on Pocket PCs are the faster graphic-methods and the better performance in thread management, which has very high overheads in J2ME. To play sound files, native methods can be easy implemented in .NET using the P/Invoke technique, the Java program needs to call a function from the Java Native Interface (JNI) for which a wrapper class is needed [12]. Java does not have many advantages when programming for windows-based Pocket PCs, especially on resource-weak devices. In fact, the interaction between the .NET CF and the underlying Windows CE APIs is a very successful combination. Java can improve its position when moving to other platforms or smaller devices where .NET CF programs are not able to run on, either because the .NET CF is not supported by the operating system or the hardware resources are too small for the .NET CF-runtime.

6.2 Coping with Packet Loss

As mentioned before, playing in ad-hoc-mode (the stations communicate directly with each other) and in infrastructure-mode (an access-point is used) is supported by the game. Due to the performance overhead of TCP [8] the User Datagram Protocol (UDP) is used, which does not guarantee a valid transfer of the sent packets. For most of the information a retransmission of lost packets is not useful, because it would influence the game-play negatively. During a rally the loss of single packets is no big problem, the opposite player will only recognize a small jump of the ball or the opposite racket. If this packet would be retransmitted, the ball and the racket would be repositioned to the values they had at the time the packet was sent originally, which would not make much sense. However, there are some packets where the transmission must be guaranteed, e.g. the information that a ball has gone out. These packets use acknowledgements to assure the correct transmission.

We now compare three strategies we used to respond to packet losses. First, every packet is only *sent once* (except the packets that must arrive), so we do nothing if a packet is lost. Second, in the *adaptive* strategy, the reaction to a packet-loss is *sending the next ten packets twice*. If a doubly-sent packet gets lost, we restart sending the next ten packets twice. Third, *all packets are sent twice*. We can measure four different probabilities, but they can not be applied to all three strategies.

- $P_1 = P(\text{Packet lost})$: Probability, that a packet is lost.
- $P_2 = P(\text{Any of doublepacket lost})$: Probability, that a packet belonging to a pair of a doubly-sent packet is lost. This in essence is like P_1, but computed only for doubly-sent packets.
- $P_3 = P(\text{Packet } n+1 \text{ lost} \mid \text{Packet } n \text{ lost})$: Probability that the second of a doubly-sent packet is lost, conditional on the presumption that the first packet is lost.
- $P_4 = P(\text{Information lost})$: Probability that the information of the packet is lost.

The most important probability is P_4, for useful strategies P_4 is zero or near zero.

6.3 Evaluation of the Strategies

The first strategy (Figure 11) can only measure the probabilities P_1 and P_4, which are equal here. As no packets are sent twice, P_2 and P_3 are undefined in this case. As suspected, most information is lost here, since if a packet is lost, the whole information is lost.

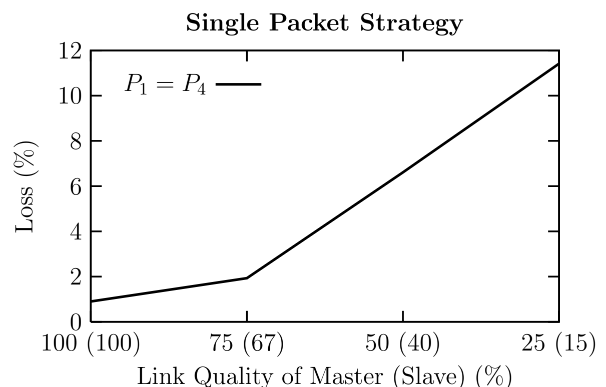

Figure 11: Loss probability of single packet strategy.

The second strategy (Figure 12) sends the next ten packets twice after a packet has been lost. Although many of the doubly-sent packets are lost (P_2), the information-loss P_4 is much smaller

now, since the conditional loss probability P_3 for doubly-sent packets is surprisingly small. This indicates that packet loss bursts are better characterized by a temporarily higher packet loss probability (here P_2 for the adaptive strategy), as for instance is used in [10], compared to models where all packets of a burst are lost [20]. Overall, this approach does not reach the values of the third strategy, but for good connections the information loss P_4 is quite low. Considering the performance-savings of this strategy compared to strategy three, it is a useful tactic for resource limited devices.

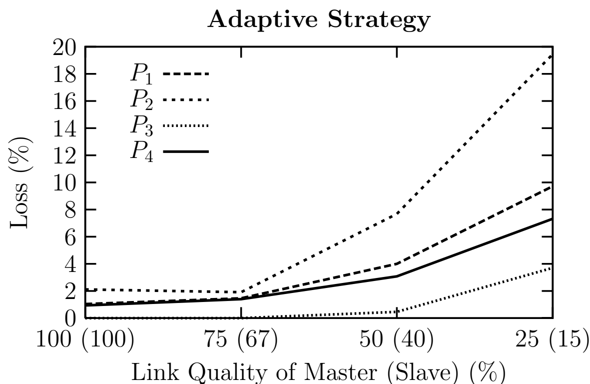

Figure 12: Loss probability of adaptive strategy.

Strategy three (Figure 13) requires the highest performance, but shows the best results. Even for a very bad link quality of about 25%, more than 97% of the sent information arrives. For a powerful device, this strategy (or an enhancement of sending every packet *n* times) seems to be the most successful.

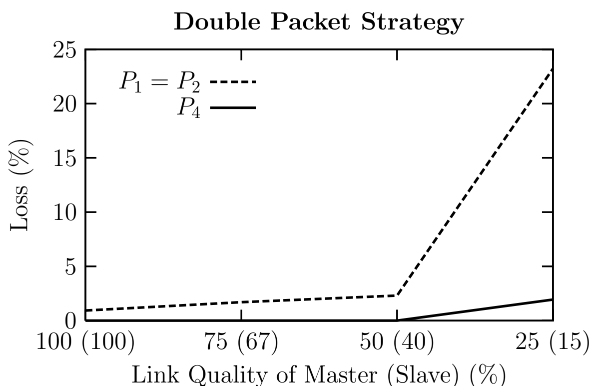

Figure 13: Loss probability of double packet strategy.

7. CONCLUSIONS

In this paper we have investigated the performance of two different PDAs and the two different programming frameworks J2ME and .NET CF with respect to their basic capabilities and computing performance.

Additionally, we demonstrated the dependence of the J2ME UDP communication on the number of packets sent per second. This indicates that not the WLAN connection but rather the low computational capabilities of the devices limit the communication performance.

Furthermore, as a simple prototypical example, we have developed a simple interactive action game for both J2ME and .NET CF which requires dependable, real-time and low-latency communication between mobile devices. For this game we have demonstrated that a simple strategy of sending packets twice can lower the negative influence of WLAN packet losses significantly.

8. REFERENCES

[1] Pocket PC benchmarks, http://www.benchmarkhq.ru/english.html?/ be_ppc.html

[2] M. Busse, B. Lamparter, M. Mauve, W. Effelsberg, Lightweight QoS-Support for Networked Mobile Gaming, in Proceedings of NETGAMES 2004, 2004, pp. 85-92.

[3] H. Hlavacs, M. Haddad, C. Lafouge, D. Kaplan, J. Ribeiro, The CODIS Content Delivery Network, in Computer Networks 48, 2005, pp. 75-89.

[4] DirectPlay, http://msdn.microsoft.com/library/ default.asp?url=/library/en-us/dnroad/html/ road12122001.asp

[5] F. Fitzek, G. Schults, M. Reisslein, System Architecture for Billing of Multi-Player Games in Wireless Environments using GSM/UMTS and WLAN Services, in Proceedings of NETGAMES 2002,, 2002, pp. 58-64.

[6] F. Fitzek, L. Badia, M. Zorzi, G. Schulte, P. Seeling, T. Henderson, Mobility and Stability Evaluation Using Multi-Player Games, in Proceedings of NETGAMES 2003, 2003, pp. 77-87.

[7] Pocket PC games, http://www.pdarcade.com/ or http://www.codeproject.com/netcf/

[8] J.F. Kurose, K.W. Ross, Computer Networking – A Top-down Approach Featuring the Internet, 2nd ed., Addison-Wesley, 2003.

[9] Terraplay Move NetSDK, http://www.terraplay.com/

[10] ITU SG 12, Contribution 104, Modelling Burst Packet Loss within the E Model, Geneva, Jan. 2003.

[11] Infinite Ventures: Lands of Shadowgate (LOS), http://www.infiniteventures.com/los.shtml

[12] Java Native Interface, http://java.sun.com/j2se/1.3/docs/guide/jni/

[13] J2ME CDC White Paper, http://java.sun.com/products/cdc/wp/cdc-whitepaper.pdf

[14] J2ME CLDC HotSpot Implementation Virtual Machine White Paper, http://java.sun.com/j2me/docs/pdf/CLDC-HI_whitepaper-February_2005.pdf

[15] Microsoft, http://www.microsoft.com

[16] Microsoft, .NET CF: http://msdn.microsoft.com/smartclient/ understanding/netcf/

[17] Microsoft Pocket PC Game API, http://msdn.microsoft.com/mobility/

default.aspx?pull=/library/en-us/dnnetcomp/html/WrapGAPI1.asp

[18] S.M. Riera, O. Wellnitz, L. Wolf, A Zone-based Gaming Architecture for Ad-Hoc Networks, in Proceedings of NETGAMES 2003, 2003, pp. 72-76.

[19] H. Ritter, T. Voigt, M. Tian, Experiences Using a Dual Wireless Technology Infrastructure to Support Ad-hoc Multiplayer Games, in Proceedings of NETGAMES 2003, 2003, pp. 101-105.

[20] H. Sanneck, G. Carle, A framework model for packet loss metrics based on loss runlengths, in Proceedings of the SPIE/ACM SIGMM Multimedia Computing and Networking Conference 2000 (MMCN 2000), pp. 177-187, San Jose, CA, 2000.

[21] IBM WebSphere Everyplace Micro Environment, http://www-306.ibm.com/ software/wireless/weme/