A Communication Middleware for Mobile and Ad-hoc Scenarios

J. Roth
University of Hagen
58084 Hagen
Germany

Abstract -- Developing software for mobile or ad hoc scenarios is very cost intensive. Different software and hardware platforms (e.g., digital cameras, PDAs, electronic pens, mobile phones) are linked together via different communication technologies (e.g., Infrared, Wireless LANs or cell phone networks). Often, mobile devices co-operate with desktop computers, thus communication links between mobile and traditional infrastructures have to be considered as well. Currently, the market for devices and communication technologies is rapidly changing. To be able to reuse software, even when the underlying communication infrastructure is modified or exchanged, applications should not be developed 'from scratch' but with the help of middleware platforms, which separate network related functions from the application. In this paper, we present the Network Kernel Framework (NKF), a middleware framework for small devices in mobile environments. Developers who want to integrate NKF support for new devices only have to realize an appropriate subset of services, thus NKF is suitable for devices with small memory and low computational power.

1. Introduction

At a first glance, realizing mobile or ad hoc connected applications currently seems to be very simple: mobile hardware is highly available and inexpensive; the Internet Protocol (IP), often viewed as the *lingua franca* of networking, connects different computers and devices across platform borders; with the help of higher level services such as Java RMI or CORBA, an application developer can easily develop networked applications by using powerful programming abstractions, first and foremost the remote method call.

Having a closer look however, the world is not so well organized and simple. There are several problems, an application developer still has to face:

- *Serious hardware limitations*: some devices (e.g. mobile phones, embedded systems) have currently not enough computational power to support programming languages such as Java. Higher-level frameworks often need a large amount of valuable memory, which, in contrast to desktop computers, is a bottleneck for mobile devices. Even if fast processors and big memories are available in principle, they consume a great amount of valuable battery power. Unless battery technology will improve significantly, mobile devices will always be far behind the capabilities of desktop computers [5].
- *Missing IP support*: for some network infrastructures or devices, IP support is inadequate or unavailable. Having, e.g., an infrared link via IrDA [9], IP support is only available with some serious drawbacks. In addition, some devices (e.g. CPen [4]) do not offer built-in IP support at all.
- *Missing platform independence and interoperability*: languages such as Java often do not have the desired level of platform independence. Java for small devices (J2ME [19]), e.g., is not compatible with Standard Java, thus we cannot use programs across platforms without re-implementing major parts of an application. The Jini framework [24], actually designed for ad hoc networks, requires full Java support, thus currently cannot be used on most mobile devices [23].
- *Missing support from manufacturers*: even if devices in principle are able to accommodate high-level communication platforms, such platforms often are not available for new devices. Currently, the market for mobile devices and wireless communication technology rapidly changes. Device manufacturers often do not invest the costs for an appropriate development support for devices with short life cycles. Third-party solutions often do not reach the required quality and tend to be unstable.
- *Incompatible reference models*: looking at the OSI reference model or the reference model provided by the IEEE 802 standard, we find strongly structured hierarchical layers. With these models, it is easy to exchange some layers without affecting upper layers. Unfortunately, most of the networks in ad-hoc scenarios (e.g. the Bluetooth or IrDA) bring their own reference models, which are often incompatible to existing models.

The lack of middleware platforms leads to monolithic and unstructured code. Moreover, changes in the communication infrastructure affect applications, which too often have to be re-engineered. If on the other hand, a developer wants to use middleware, but no platform is publicly available for a specific device, the entire development process is twofold: in addition to the application, the developer has to realize at least a rudimentary middleware platform.

In this paper, we present the *Network Kernel Framework* (*NKF*). On one hand, it leads to lean runtime systems, which even run on very small devices, and at the same time it separates all network related functions from the application. Moreover, the framework relieves an application from additional functions such as data encoding, compression and encryption.

2. Related work

The idea of separating network access from applications is not new; a huge variety of tools and middleware platforms supports developers of distributed applications. Well-known examples are, e.g., RPC [17], Java RMI [18], CORBA [11] and IBMs DAE [7]. These platforms significantly simplify the development of distributed applications with the help of powerful services such as remote procedure calls or remote method invocations. However, to offer this support for new devices, the entire platform has to be ported to a new hardware or operating system. This is usually a time-consuming and cost-intensive task. Even if there exists platform support for a new device, this support sometimes can be viewed as a 'black box', i.e. the creation of applications is simplified, but the platform itself cannot easily be modified. This is especially true, if third parties developed the middleware platform for commercial purposes. Usually such platforms do not come along with their source codes, thus it is difficult to integrate new security protocols, access new network infrastructures or include new data encoding schemes.

Higher services, which heavily use mechanisms such as RPC and RMI (e.g., [20]), are restricted to the specific communication paradigm (e.g. remote calls), provided by the basic platform. Often, such platforms omit a wide area of other useful communication paradigms. It is not possible to map mechanisms such as multicast or anycast on network level directly to remote calls. This forces applications to inefficiently use 'loops' of unicast instead of using native multicast capabilities.

Some systems provide limited access for integrating different network capabilities into existing platforms. Microsoft Windows uses abstract sockets, so-called *winsocks*, which either can be TCP/IP, IPX/SPX or IrDA sockets [9]. Applications using winsocks can use different communication infrastructures without too many changes. Nevertheless, applications cannot be fully network independent, since there are slightly different addressing schemes and discovery mechanisms for different networks. Similar to winsocks, Java offers so-called *socket factories*, which give developers the opportunity to integrate new kinds of network connections into an application. Since original Java sockets base on the TCP/IP protocol suite, new sockets must have similar characteristics.

Other approaches avoid network integration issues and only define a high-level protocol, thus leave it to the developers or operating system to make the actual implementation. *OBEX* (*object exchange protocol*) [8] is a protocol for exchanging arbitrary objects between different devices such as digital cameras, PDAs and mobile phones. The current OBEX protocol, better known as the 'beaming' protocol [2], uses the IrDA infrared stack for communication, but in principle can run on every reliable network transport layer such as Bluetooth or TCP. *SOAP* (*simple object access protocol*) *follows a* similar concept [21]. With SOAP we can exchange arbitrary application objects using XML for data encoding. Since various platforms can interpret XML, SOAP is highly flexible and interoperable. SOAP uses HTTP to exchange messages, thus is limited to TCP network connections.

To sum up, there exist different levels of development support for distributed applications. Higher-level platforms offer powerful services, but request high implementation efforts to offer this kind of support for new hardware and software platforms. Lower-level support can be realized much easier on any device, but leave much work for the developer. The problem is to find an appropriate balance between development support and integration efforts.

3. The Network Kernel Framework

The Network Kernel Framework (NKF) is a middleware platform for small devices such as PDAs or digital cameras, as well as for traditional desktop systems. It is especially designed for supporting new devices, which do not come along with publicly available middleware platforms such as CORBA or RMI. In such cases, developers have to implement the middleware in addition to the actual application, thus NKF is easy to realize and does not make high demands on target platforms. Though we implemented NKF on Java and C, NKF does not rely on a specific programming language paradigm. In the following, we present NKF in more detail.

3.1. Protocols, Modules, and Methods

Fig. 1 shows an overview of NKF's architecture. The framework acts as an intermediate layer between lower-level services provided by the operating system and higher layers such as applications or high-level services. The framework consists of two different kinds of modules: framework and kernel modules. Four *framework modules* are fixed parts of the framework, each offering a specific service:

- The *lookup module* looks up other communication parties inside the network.
- Once the lookup module has found a service, the *negotiation module* negotiates parameters for a specific connection, e.g. which protocol to use and how to present data.
- The *service module* performs additional set up and configuration functions for special network connec-

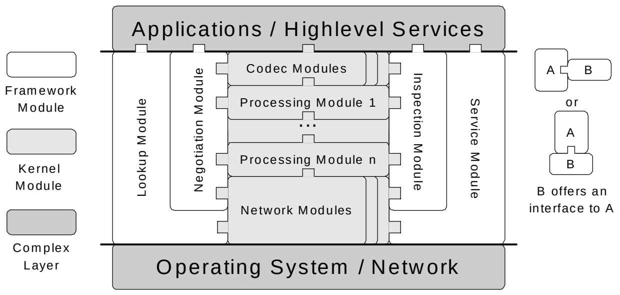

Figure 1: The NKF architecture

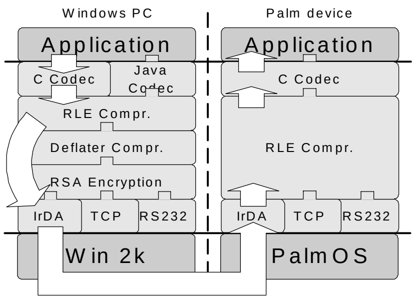

Figure 2: NKF data flow example

tions. For a serial connection, e.g., parameters such as baud rate and parity have to be configured.

- With the help of the *inspection module*, higher services can look up properties of a specific connection. Using, e.g., a cellular phone connection, it may be helpful to query the current receiver input level.

In contrast to framework modules, *kernel modules* may vary in number and specific functionality. We distinguish three types of kernel modules:

Network modules offer a uniform interface to lower-level network services. Network modules are, e.g., TCP, UDP, MulticastIP, IrDA, Bluetooth or RS232 modules. We describe network modules and their interfaces to upper modules in more detail later.

Before delivering data to the network, *processing modules* pre-process data packets. To save network bandwidth, packets may, e.g., be compressed via a deflating algorithm. Processing modules can be piled up: after compressing data, e.g., the result can be encrypted with asymmetric encryption against unauthorized listeners. On the receiver's site, the same stack of processing modules has to exist.

Codec modules encode application data for transportation. Same codec modules use the same coding across different platforms and programming languages, regardless of the specific internal data format. Application developers have not to struggle with, e.g., string termination, integer size or byte ordering. Examples for codec modules are: C codec (C style data format), Java codec (Java data stream format) or String codec (every data transferred as Unicode string). Note that, e.g., the Java codec may be available for other programming languages than Java - it determines the transportation format rather than the interface language to the application.

We give an example to clarify this concept: suppose two applications want to communicate, one running on a traditional Windows PC, one on a Palm handheld device (see Fig. 2). We connect both computers either via a serial cable, via infrared (IrDA) or via the TCP/IP stack (which itself either uses a serial cable or infrared). Both computers are equipped with a number of processing and codec modules. The 'weaker' device, the Palm, offers only the very simple compression algorithm RLE (run length encoding) and a C style data coding, the Windows PC offers in addition the more complex deflater compression, an asymmetric encryption and Java style data encoding. To connect, the devices have to choose modules, which are available on both platforms, i.e. the C coding, RLE compression and the IrDA network module. The set of modules can either be specified by the corresponding application or can be negotiated automatically. We describe the latter case in more detail later.

Once connected, both applications can simply send and receive data without considering the underlying infrastructure (Table 1). If in the future, new network, processing or codec modules are integrated, they can simply be added without adapting existing applications.

The Windows PC is the initiator of the communication (`new Connection`) and the Palm waits for incoming connections (`NKF_waitForConnection`). Once a connection has been established, both parties can send and receive messages (`send` and `receive` statements). A single message consists of one or more data entries, e.g. strings or integers. A developer uses `putX` statements to add data to an outgoing message and `getX` statements to read data from a received message. After sending and receiving messages, both parties terminate the connections (`disconnect`). To sum up, this example outlines the following benefits of NKF:

- The two applications communicate across hardware platform borders (Windows PC ⇔ Palm device).
- The two applications communicate across program-

Table 1: Source code examples

Java / Windows PC	C / Palm device
conn=new Connection ("irda://palm:2000/RLE/C");	conn=NKF_waitForConnection ("irda://:2000/RLE/C");
	NKF_receive(conn);
conn.putInt(123);	i=NKF_getInt(conn);
conn.putString("hello");	s=NKF_getString(conn);
conn.putBoolean(true);	b=NKF_getBoolean(conn);
conn.send();	
conn.receive();	
i:=conn.getInt();	NKF_putInt(conn,456);
s:=conn.getString();	NKF_putString(conn,"hello");
b:=conn.getBoolean();	NKF_putBoolean(conn,0);
	NKF_send(conn);
conn.disconnect();	NKF_disconnect(conn);

ming paradigms (object oriented ⇔ traditional imperative) and language borders (Java ⇔ C).

- The two applications communicate regardless the underlying communication technology, security protocols or data compression algorithms.

One weak point of this example is the string "irda://..." passed to the opening procedures at the beginning of the applications. This string is obviously not independent from the underlying communication infrastructure. In a later section, we describe mechanisms to avoid such strings.

3.2. Network Modules

Network modules are the most important NKF modules. With network modules we can change the underlying network without affecting processing or codec modules. Applications only have to be adapted, if they make explicit use of specific network capabilities via the service or inspection module.

To offer a wide variety of network functions to upper layers, network modules must have an interface to integrate a big number of network protocols such as

- TCP/IP suite protocols (TCP, UDP or Multicast IP);
- a reliable multicast protocol based on Multicast IP and UDP:
- infrared protocols based on IrDA;
- Bluetooth;
- GSM or UMTS, and
- protocols, which communicate via a USB, Firewire, or via a serial or parallel port.

For debugging and testing, an additional network module for simulation is useful. This module can act as a usual network module, but performs configurable bandwidth reduction and delays, thus is ideal for testing an application under realistic conditions.

Different network modules have different characteristics. A module can support stream connections or datagrams. If datagrams are used, the module can support unicast, multicast or anycast; properties concerning packet loss and packet ordering may be different. To support different communication capabilities, we defined a number of *methods*, each specifying a specific communication scheme, e.g.:

- `STREAM`: stream connection (e.g.TCP, IrDA, serial),
- `UNICAST`: unreliable datagrams (e.g. UDP),
- `REL_MULTICAST`: reliable multicast
- `ANYCAST`: unicast to one of a group (available in IPv6, currently emulated).

Usually, a network module does not support all of the above methods. Via the inspection module, an application can query for methods a module actually supports; a result can be one of {`NOT_SUPPORTED`, `SUPPORTED`, `EMULATED`}. We use the `EMULATED` entry, if a specific network module does not support a method in a native manner. A UDP module, e.g., can emulate multicast using unicast in a loop. This is, of course, much more inefficient than using native multicast. Nevertheless, if an application uses the multicast method directly rather than making a loop of its own, we can install an optimized network module later without changing the application.

3.3. Security

Security is an important issue in mobile and ad hoc scenarios. A communication infrastructure should address the following aspects *Privacy, Authenticity, and Integrity.* To ensure security, current mobile network protocols use cryptographic algorithms such as asymmetric and symmetric encryption and hash functions. Unfortunately, cryptographic algorithms, especially asymmetric encryption algorithms, need a lot of computational power, and thus in some protocols security is not supported. OBEX [8], e.g., only provides authentication on session level. The stronger message-level authentication is not available. More critical, OBEX ensures neither privacy nor integrity. Other mobile protocols only use symmetric encryption and leave it to the application to generate keys via, e.g., public key exchange mechanisms.

An application developer may want to have fine-grained control about the level of security provided by the network. For some applications, no or only low-level security may be sufficient, whereas other applications require higher levels of security. NKF offers the desired flexibility:

- Demanded security functions can easily be integrated into an NKF stack with the help of processing modules.
- The negotiation module can initiate key exchange and session authentication.
- Processing modules can perform authentication on message-level by adding message authentication codes (MACs) to outgoing messages and verifying MACs when receiving messages.
- The NKF stack may rely on security functions provided by the network. If, e.g., the network provides *Transport Layer Security* (*TSL*) [6], no additional security modules inside NKF are necessary.

A developer may adjust the level of security without affecting the interface to the application. In other words: a developer may develop an application without considering security issues and may add security algorithms later. Currently, many devices are not able to offer complex security functions. If, in the future, new algorithms are available in hardware (e.g. encryption chips, special smart cards with security algorithms), security functions can easily be integrated into an existing application.

3.4. Encoding and Decoding Data

To exchange data across operating system and hardware borders, data have to be encoded in a platform independent manner. There exists a variety of solutions to address this issue:

- Data format specifications for native data types (the CORBA approach).
- Equivalent virtual machines on different devices (the Java approach).
- XML to encode data (the SOAP approach).
- MIME types [3] (the OBEX approach).

All these solutions have advantages and disadvantages. Some applications only transfer small amounts of data, thus some of the solutions above are strongly oversized. If, e.g., an application only transfers unstructured data types such as integers and strings, an XML parser or Java virtual machine are not adequate, specifying byte ordering and string termination are sufficient.

With NKF, a developer can realize all the mechanisms described above; it offers the flexibility to both, either use high-level concepts such as XML or MIME types, or lean mechanisms. To realize an NKF stack for a specific platform, a developer can choose an appropriate level of support by creating a set of codec modules, which contain the appropriate pairs of `get` and `set` routines. An application can request specific modules at runtime or leave it to the negotiation module to negotiate an appropriate codec module with the communicating party.

3.5. Identifying Hosts, Devices, and Services

Different network infrastructures use different naming conventions to address hosts and services. Since the interfaces of all network modules have to be identical, the address format has to be identical as well. We use the simple addressing format (host, channel), where *host* is a string specification of the host address and *channel* is a number specifying a service on a host. This scheme is appropriate for most networks. TCP/IP hosts, e.g., are identified by the IP address or a host name, which both could be presented as a string. A port number presents a specific service on a host. In the same way, we identify IrDA devices by a device name. Services on a device are distinguished by a number called the *Link Service Access Point number* (*LSAP*). Exceptions of this scheme exist, but can easily be adapted: a serial connection, e.g., does not distinguish between different services on one device. Nevertheless, our solution of the serial network module implements its own scheme of service numbers with the help of a simple protocol.

For host names, we use two fundamental different namings: A *global naming* has a world-wide unique name for a specific host or device. A *local naming* uses different names for the same host or device. A name is only unique for a specific other host. Examples for global namings are TCP and UDP: using a fully qualified Internet name, a host has a worldwide unique address. An infrared connection based on IrDA uses local naming. For a specific host, an infrared device in range may have a name `A`, where another host has no idea about `A` because it is not in infrared range. Even worse, both hosts may know different devices with the same name `B`.

The network module specifies the corresponding naming scheme; an application can query the scheme using the inspection module. The difference between local and global names is important when distributing host names to other devices: another device only can use global names directly. To access devices with local names, we use a special process, the *communication gateway*. Communication gateways run on top of the NKF platform and allow a device to address other devices via their local name. For this, a so-called *global cascading name* is used. We define global cascading names as follows:

```
c_global_name:
    global_name
    local_name,c_global_name
```

where `global_name` and `local_name` are local or global device names. The string

`local_name`$_1$`, local_name`$_2$`...,`
`local_name`$_n$`, global_name`

addresses a specific device with local name `local_name`$_1$, which is connected to a device `local_name`$_2$, which in turn is connected to device `local_name`$_3$ etc. In order to get a global unique address, the last device name has to be a global name. Usual cascading names have not more than two components. E.g, `CPen,132.176.67.44` addresses the device 'CPen', which is connected to a TCP host with its corresponding IP address. Global cascading names are passed to gateway processes, which serve as a kind of routers and link different networks together. Gateway processes are not specified as components *inside* the network kernel framework, but viewed as higher-level servers built *upon* NKF.

3.6. Specifying Connection Properties

Usually, an application looks up a service via the lookup module. When an appropriate connection is established, a reference (i.e. an object handle or pointer) representing the connection is passed to the application. The application uses the reference for sending and receiving data; usually it is not required to configure the connection. In some cases however, an application may want to have explicit control over the used modules. To conveniently specify network, processing and codec modules, we use a notation similar to web URLs:

`protocol://target:channel/proc`$_1$`/...`
`/proc`$_n$`/codec#method`

where target can be one of {`<emtpy>`, `host`, [`host`$_1$,... ,`host`$_m$]}. Empty targets are used when a host specifies a server end point for incoming connections. With a list of hosts we address multiple hosts at a time for, e.g., multicast. Some parts of the string, e.g., the method, can be left out, if either there exist default values or the parameters are negotiated at runtime. Example strings are:

- `tcp://carmen:5555`: a TCP connection to host carmen, port 5555, processing and codec will be negotiated;
- `irda://palm/RSA/deflater/Java`: an infrared connection using RSA encryption, data compression and Java data encoding.

These strings are passed to a connect operation to establish the desired connection. With the help of this notation, an application developer can simply test different connection types in an application by just adapting a single string.

3.7. Discovery, Lookup, and Negotiation

Passing a predefined string to an application is suitable for testing scenarios. However, if a device connects to an unknown environment, it knows neither other host addresses nor available modules on peer sites. Applications must have tools to discover the network environment and negotiate communication properties at runtime. For this, we use three mechanisms inside the framework:

Discovery is performed by network modules and returns the address of other accessible hosts and devices. A discovery of a TCP module, e.g., returns all hosts of a subnet; a discovery of the IrDA module returns all devices, which are in infrared range. The discovery mechanism is the most important function for ad hoc connected devices. It highly depends on the module and can be released in many ways. The IrDA protocol stack, e.g., has a built-in discovery mechanism, thus discovery is available without extra costs. Other protocols, e.g. TCP/IP, may need additional functions to discover available devices. A module can ask network devices such as routers or switches for the corresponding information or send discovery messages via native multicast. Regardless which strategy we use to discover the network, the corresponding mechanisms are encapsulated inside the network module.

Lookup: once having a list of hosts, an application can look for specific services. Lookup is performed by the lookup module and makes use of a protocol running over well-known channels. An application either can retrieve a complete list of services available or can specify a query. Each service is specified by a name, a version number and an instance number. With the latter, identical services on a single device can be distinguished. If native multicast modules are available, we use them both, for discovery and lookup.

Negotiation: when an application decides to connect to a service, the actual connection properties have to be negotiated. For this, the negotiation module negotiates the network module, the stack of processing modules and the codec module. An application can pass a list of preferences to the negotiation module, which maps modules to one of {REQUEST, PREFER, EVADE, REFUSE}. This however can cause a connection operation to fail, if requested modules are not available on peer sites. Thus, a developer has to use this feature carefully.

After the module stack is negotiated, the negotiation module asks the individual modules for module-specific negotiation. An encryption module, e.g. may want to exchange public keys with peer devices. Other modules negotiate runtime parameters. Since this kind of negotiation is highly module-dependent, the modules themselves are responsible for it.

3.8. NKF Support for Very Small Devices

One could argue that building a complete stack of modules is a lot of work and may not be appropriate for mobile devices such as cell phones, wristwatches or PDAs. An example for a lean but fully operable NKF stack for such devices has one codec, one network and one service module. Since these modules are fixed, negotiation and inspection modules are not needed. Assuming that the device directly communicates to another serial device, the lookup module is useless as well. Merely the service module is necessary to configure serial parameters such as baud rate, parity etc. This NKF stack is easy to implement and costs only a few hundred lines of code, but an application developer can already benefit from the concept, since the stack can be extended later without changing the application.

3.9. Interoperability with Existing Protocols

Connecting solely NKF-enabled devices is a rare case. Only if all devices and computers for a specific application can be equipped with an NKF stack, we can use the described NKF lookup and negotiation features. Sometimes however, we have to be interoperable with existing devices and network protocols, which are not under control by the application developers. To, e.g., connect to a traditional web server, a device has to use HTTP via TCP/IP. Common web servers are not able to handle, e.g., negotiation requests from NKF devices.

NKF can be used in co-existence with existing protocol in two ways. An NKF-enabled device can simply connect to, e.g., a TCP/IP device. For this, it has to be configured as follows:

- no negotiation and lookup modules,
- no processing module and a *null* codec (a codec, which passes bytes to the underlying modules without modifications),
- fixed network module (e.g., TCP).

We call such a configuration the *transparent mode*. Stacks in transparent mode can interoperate with other network stacks, but we give up some advantages of the NKF concept such as the negotiation facility. However, using NKF in transparent mode, we still benefit from the network independent development style of NKF.

The second way is much more complex. It follows an idea, presented in [1]. An architecture called *split connection* was originally introduced to solve some transport layer problems in wireless environments. In this scenario, a mobile device is connected to a base station via wireless link. The base station has a wired TCP/IP connection to a stationary host.

With this architecture it is possible, to, e.g., connect a mobile device to a traditional web server without changing the server application. The mobile device is equipped with full NKF capability, thus can connect to a base station via several protocols (e.g., Bluetooth, IrDA, WLAN). The base station works as an intermediate host, converting any kind of network into TCP/IP.

The split connection architecture combines the advantages of NKF with the TCP/IP interoperability. Often, base stations are already available for a special communication infrastructure (e.g. WLAN, GSM), thus no additional costs result from setting up the specific gateway application.

3.10. Ad-hoc Routing

Using mobile devices without the help of stationary base stations, we have to address the ad-hoc routing problem. There exists a huge variety of routing protocols, e.g., DSDV [13] or TORA [12], which allow the mobile devices to connect in an ad-hoc manner without the need of a central administration. However, for a specific realization, we have to face two problems:

- Even if the routing algorithms offer a general solution for arbitrary networks, an implementation is tailored to a specific network stack. Once the development has been finished, it is difficult to use the routing algorithm in other network scenarios without serious re-implementations.
- So-called *overlay networks* [10] use more than one network stack inside the same logical network. There exist protocols, which make use of such infrastructures (e.g. [22]). As a major drawback, developers have to access more than one network stack inside the same algorithm, which results in additional development costs.

NKF can help to overcome these problems. With NKF, a developer can develop ad-hoc routing algorithms in a modular and network-independent manner. Once a specific algorithm has been realized, we can easily re-use it in other network scenarios without re-engineering.

Using the simulation module, a developer can test a specific ad-hoc protocol on one computer, without to set up a wireless network. Setting up a wireless network with mobile devices is very cost-intensive. Inside the simulation environment, we can test scenarios, which in reality would be very difficult to prepare. Once a specific algorithm is tested and debugged inside the simulation environment, we can use it without any changes in a real environment.

3.11. Higher-level Services

There are many services conceivable, which are currently not part of the Network Kernel Framework such as trading or directory services. To keep the framework lean and suitable for small devices, it is not intended to integrate such services in the future, but the framework offers a common platform for construct such services on top of the platform. Third-party developers may develop, e.g., their own directory services on NKF, which then are highly network independent and portable.

Another service intentionally missing is the remote method invocation or remote procedure call. To access objects, which have been developed in different languages, remote calls either require homogenous language support (e.g. Java for RMI) or high-level interfaces (e.g. IDL for CORBA). Both concepts are, in our opinion, not suitable for small computers in heterogeneous environments. Nevertheless, NKF can act as a framework for realizing such services if required. If, e.g., all communication peers run Java, it is easy to implement on top of NKF a remote method invocation protocol, which is much more network independent and flexible than the original RMI.

3.12. Development Issues

As discussed above, developing distributed applications in mobile and ad hoc scenarios often has two aspects. If target devices do not come along with a powerful communication platform, we have both to develop the actual application *and* the communication middleware. The ongoing work on NKF is twofold a well. On one hand, we have to provide a sufficient set of NKF sample stacks; on the other hand, we need 'killer applications' to verify the approach.

We offer support for TCP/IP suite protocols, IrDA and serial connections. Supported platforms are desktop systems with Windows, handheld devices with Windows CE and PalmOS. We currently work on NKF support for the CPen based on the ARIPOS [4] operating system.

We verified NKF with the help of two realistic applications. We use a first version of NKF in our groupware platform *DreamTeam* [14], which acts as a common platform for distributed applications such as shared whiteboards and shared text editors. DreamTeam was originally created to run on desktop systems, but, with the help of NKF, we are currently working on a DreamTeam extension for palm devices.

We designed the second platform, *QuickStep* ([15], [16]), especially for handhelds such as PalmOS or Windows CE devices. With QuickStep, users can share application data, e.g., appointments, addresses, memos or business cards with other users. QuickStep has much more flexibility than the 'beaming' protocol OBEX. Users can share entire databases and do not have to transfer data record by record. Moreover, connected users do not have to be 'in range', i.e. users can share data even when they reside in different buildings. For this, hierarchical connected servers act as relays between end users.

When developing QuickStep, we faced the following issues:

- Parts of QuickStep are running on traditional workstations, others are running on handhelds, Thus different operating systems and different programming languages got involved.
- Communication links between handhelds and PCs are wireless where links between the PCs base on traditional networks.
- For the handheld devices, no appropriate middleware platform was available.

Due to the heterogeneity of devices and networks, QuickStep was an ideal application for verifying NKF. We developped a number of sample applications with QuickStep, e.g. a calendar tool, which allows members of a meeting to schedule appointments for future meetings. In addition, we developed a brainstorming tool and a business card collector.

4. Conclusion and future work

In this paper, we presented the Network Kernel Framework NKF, a platform for distributed applications in mobile and ad hoc environments. The NKF approach is especially useful for new devices, for which traditional middleware solutions are not available. The framework is modular; for a specific device a developer may implement only an appropriate subset of modules. NKF can easily be implemented on even small devices. Once an application uses only framework services to access the network, modules for new communication infrastructures, new compression or encryption algorithms and encoding schemes can easily be integrated into the framework without affecting existing applications. This saves implementation costs.

To verify the approach, we built two platforms on top of NKF: DreamTeam and QuickStep. Especially QuickStep heavily benefits from NKF since it runs both on handheld devices as well as on traditional workstation and can use various network infrastructures.

In the future, we will go into two directions. First, we want to significantly increase the number of supported networks and runtime platforms. Secondly, we want to create high-level services such as communication gateways and service traders on top of NKF. Offering sufficient platform support as well as high-level services, an application developer significantly benefits from the NKF concept.

References

[1] A. Bakre, R. R. Badrinath, *I-TCP: Indirect TCP for mobile hosts*, 15th Int. Conference on Distributed Computing Systems, 1995

[2] C. Bey, E. Freeman, D. Mulder, J. Ostrem, *Palm OS SDK reference*, 3Com, http://www.palm.com/devzone/index.html

[3] N. Borenstein, N. Freed, *MIME (Multipurpose Internet Mail Extensions) part one: Mechanisms for specifying and describing the format of Internet message bodies*, RFC 1341, Sept. 1993

[4] C-Technologies, *C-Pen SDK Version 1.0*

[5] D. C. Cox, *Wireless personal communications: A perspective*, in the *Mobile Communications Handbook*, Second Edition, Gibson J. (ed), CRC Press and Springer Verlag, 1999

[6] T. Dierks, C. Allen, *The TLS protocol*, RFC 2246, Jan. 1999

[7] IBM, *Distributed Application Environment - Communications system application programming Volume 1, 2, and 3*, Document Number SC28-8278-05, Sixth Edition, June 1994

[8] Infrared Data Association, *IrDA Object exchange protocol*, http://www.irda.org, March 1999

[9] A. Jones, J. Ohlund, *Network programming for Microsoft Windows*, Microsoft Press, 1999

[10] R. H. Katz, E. A. Brewer, *The case for wireless overlay networks*, in Proc. of the SPIE Multimedia and Networking Conference (MMNC ’96), Jan. 1996

[11] Object Management Group, *The Common Object Request Broker: Architecture and specification*, Revision 2.2, Document 98-07-01: Feb. 1998

[12] V. D. Park, M. S. Corson, *A highly adaptive distributed routing algorithm for mobile wireless networks*, Proc. of the IEEE INFOCOM ’97, Kobe, Japan, April 1997

[13] C. E. Perkins, P. Bhagwat, *Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers*, Proc. of the SIGCOMM ’94 Conference on Communications, Architectures Protocols and Applications, Aug. 1994, 234-244

[14] J. Roth, *'DreamTeam': A platform for synchronous collaborative applications*, AI & Society, Vol. 14, No. 1, Springer Verlag London, March 2000, 98-119

[15] J. Roth, C. Unger, *Using handheld devices in synchronous collaborative scenarios*, Second International Symposium on Handheld and Ubiquitous Computing 2000 (HUC2K), Bristol (UK), Sept. 25-27 2000, LNCS 1927, Springer, 187-199

[16] J. Roth, *Information sharing with handheld appliances*, 8th IFIP Working Conference on Engineering for Human-Computer Interaction (EHCI'01), Toronto, Canada, May 11-13 2001, LNCS 2254, Springer, 263-279

[17] Sun Microsystems, Inc., *RPC: Remote Procedure Call, Protocol specification version 2*, RFC 1057, IETF, June 1988

[18] Sun Microsystems, Inc., *Java Remote Method Invocation specification*, Oct. 1997

[19] Sun Microsystems, Inc.: *Java 2 Micro Edition SDK*, http://java.sun.com

[20] A. Schill, S. Kümmel, *Design and implementation of a support platform for distributed mobile computing*, Distributed Systems Engineering Vol. 2 No. 3, 1995 128-141

[21] SOAP Webservices resource center, http://www.soapwrc.com/

[22] M. Steenstrup, *Cluster-based networks*, in Perkins, C., E. (ed): Ad Hoc Networks, Addison Wesley, 2001

[23] T. Urnes, A. Hatlen, R. Johansen, O. Myhre, *Using mobile code to build a smart kitchen*, Personal Technologies, Vol. 4, No. 4, 2000, 202-204

[24] J. Waldo, *The Jini architecture for network-centric computing*, Communications of the ACM, Vol. 42, No. 7, July 1999, 76-82