QoS applied to security in mobile computing[*]

Terje Fallmyr and Tage Stabell-Kulø
Department of Computer Science
N-9037 University of Tromsø
NORWAY

e-mail: `terje@cs.uit.no` and `tage@acm.org`

Keywords: Mobile computing, Quality of service (QoS), adaptability, reconfiguration, notification, security, trust management.

June 30, 1997

Abstract

Hand-held mobile computers have the potential to become important communication tools for roaming users. As such, they will also become very personal. They will be used under a wide range of operating conditions, and tight user control will be enforced on issues like power consumption, consistency control, and trust management. Their ability to adapt will be the key to their success.

In this paper we outline our notion and use of Quality of Service (QoS) to the design of adaptive software systems for mobile computers. They have been developed in the MobyDick and GDD projects. We do not emphasize on the provision of QoS guarantees. In stead, our notion of QoS is used to convey relevant and timely management information between service users and providers on the correct abstraction level. It structures adaptability management in the hand held machine, and it captures adaptability to changes both stemming from the hosting environment and user commands.

As an example of how the architecture works, the importance of adaptivity of security services for personal companions are explained, and we show how our notion of QoS may realize adaptable security services.

[*] This work was supported by the MobyDick (ESPRIT LTR 20422) and GDD projects (NFR 112577/431)

1 Introduction

A *Personal Companion* is a very personal, small, portable computer and communications device. It may replace many of the items that people carry around, like cash, credit cards, keys, diary, cellular phone, and pager. It is a roaming, battery driven and relatively resource poor computer. The quality of the services it offers, as well as how they actually are executed best, will vary with the machine's current capabilities. This is to a large extent determined by its current hosting environment at any time.

As personal companions become ubiquitous [Weiser91], they also become important to the owner. They will contain the keys and information that is needed for many everyday actions. Given their importance, we must expect that users want the ability to control them to a much greater extent than is usual with ordinary desktop or notebook computers. This spans a wide range that includes control over how and when the limited power is consumed, and determining which parties to trust in transactions that require security. Therefore, they should be adaptive and work predictively under a wide range of conditions.

The personal companion's close affiliation to the personal sphere, places new and challenging demands on security. The fact that a personal companion is trusted, makes the user (as an individual) vulnerable to all sorts of attacks, ranging from theft of the machine to Trojan horses. Security must be handled with the outmost care in order to use the personal companion to purchase goods from a foreign, and possible non-friendly provider. At the same time, the security scheme must be flexible. In a setting where denial of service is not tolerated, insufficient resources may dictate that a costly but safe security scheme is traded for a less costly but less secure scheme with a greater chance of success, an example of which will be shown in Section 5.3.

In security, *trust* is difficult to manage. This is so since it is a feeling, and as such can not be measured or treated algorithmically. Therefore, any decision that relies on trust must ultimately be made by a human. This is true regardless of what "purpose" the trust plays (binding a key to a person or trusting the internal clock in a machine to show the correct time, or whatever). As humans we can draw on many sources to become confident that some statement is true. For example, based on experience, you trust your watch to show the (more or less) correct time although you can neither prove it to run correctly, nor that it has not been tampered with. Furthermore, you are confident that you will notice if it goes wild, and do not believe it will engage in a scheme to trick you into believing it is night at noon. This might seems trivial, but as we will see, trust is an important resource that can be exploited. In particular, trust is closely tied to the "quality" of any security scheme, and is thus a natural component in a system based on the notion of QoS.

Key design issues for the software system of personal companions are their ability to provide predictable service levels and behaviors in the face of varying operating conditions and user controls. Their ability to adapt, both at the application and system levels, will be key to their success. The importance of adaptability has been pointed out several places, e.g. [Davies93, Satyanarayanan94, Imielinski94, Forman94, Schilit94, Kaashoek94].

We present a software architecture that is a part of the MobyDick and GDD projects, and which takes a novel approach to meet the demands for adaptability in personal, mobile computers. Our notion of *Quality of service* (QoS) does not emphasize on the provision of QoS guarantees. In stead, QoS is used to convey non-functional service information as well as relevant and timely management information between service users and providers, and to provide it on the correct abstraction level. The architecture structures adaptability management in the mobile computer, and handles adaptability to changes stemming both from the hosting environment and from user commands.

The QoS architecture is outlined in Section 2. Section 3 explains integrated QoS-based management, while Section 4 presents the software architecture. Adaptable security is presented in Section 5. We sum up in 6.

2 A QoS architecture for adaptability

(QoS) is an attractive model for resource allocation and sharing, and has gained much attention in the transfer of multi-media documents. QoS based resource allocation is closely linked to the service model and is based on service users requesting a resource or service on some level of quality from a service provider.

2.1 Quality of service in statically connected systems

In statically connected systems, the service provider normally verifies (end-to-end) if the request can be granted. If the service provider grants the request (possibly after negotiation), the two parties enter a *QoS contract* that gives the service user a “guarantee” that the service level stated in the contract shall be sustained.

On a long time scale, the service provider may apply admission control to limit resource usage to fulfill the obligations in the existing contracts. On a short term scale, fluctuations may be handled by the transport channel [Huard96]. Service users may normally rely on the guarantee, and that they are being granted the resources stated in the contract until they release them. This model relies on the stability of the service provider and the resources and services it requires.

2.2 Quality of service in mobile systems

Availability and quality of resources in a mobile computer are in general unpredictable, so a service provider in a mobile computer cannot assume the same kind of stability as its peer in a static system. Therefore, it cannot normally issue QoS guarantees that a service user can rely on.[1] QoS-based resource management in a mobile computer must take this fundamental difference into account.

As QoS-based approaches for static systems focus on the provision of guarantees, they tend to pay less attention to the situations that occur when the contract actually is violated by the service provider for some reason. A roaming computer, however, will often enter not foreseeable situations and new environments, In some of them the current QoS level in one or more contracts cannot be sustained. The user may also decide to turn the machine off, or issue a control action that suddenly alters the way resources are allocated throughout the system. These situations are not errors, but are modus operandi for mobile computers.

Given sufficient resources, the software entities in our architecture request and release resources as in statically connected systems. But when resources degrade, the

[1]There are exceptions, like when the mobile computer is well connected, and behaves like a small ordinary computer.

system will exhibit a potentially high portion of resource revocations that cannot be anticipated either by service users or providers. Hence, we do not emphasize on the provision of QoS guarantees. In stead, our notion of QoS is designed to convey relevant and timely management information between service users and providers, and on ensuring that it is presented at the correct abstraction level.

2.3 The building blocks

The software system of the personal companion is composed of modules whose internal activity is not externally visible or accessible. Each module contains both service and management operations.

Qualitative aspects associated with a sequence of operations—e.g. security level, consistency requirements, or data stream quality—are expressed in terms of QoS parameters. The sequence of operations may for instance pertain to a network connection (from `connect` to `disconnect`), or operations on a file (from `open` to `close`), or a transaction.

The first operation in the sequence will trigger a negotiation phase during which the invokee will request (end-to-end) the necessary contracts and resources it needs. When the two modules agree on which QoS parameter values that is to be supported by the invokee, they have a QoS contract.

In an adaptive system, the invoker needs the possibility to get information about relevant states and events concerning ongoing service operations that is normally considered internal to the invokee. This may happen asynchronously with respect to the state of the invoker's current underlying computation or the ongoing service operation. The set of management operations are used for this. It contains operations for the exchange of notifications and control operations between the invoker and the invokee, and for the invoker to fetch relevant management information from the invokee. They give the invoker timely and controlled access to invokee-intrinsic information when needed and according to the agreement stated in the QoS contract between the two parties. The QoS contract states what information the invoker had requested. Thereby both scope and abstraction level of the information is limited to the invoker's needs and abilities.

2.4 The QoS contract

A QoS contract in the architecture consists of several parts, including a *contract type*, a set of *QoS parameters*, a *contract identifier* (CID) and a *monitoring method*.

The contract type identifies what "topic" the contract is about (e.g. multi-media, audio, video, security, consistency) and makes it possible for adaptability management to take actions that are suited for that type, and to make priorities between types. For instance, telephony may get higher priority than file transfer or e-mail.

Each QoS parameter consists of a value pair, one representing the low end and the other the high end, of what is called a *tolerable range* (the two values may be equal). As long as the delivered QoS parameter values during service stay within the tolerable range, the provider is said to behave according to the contract. When a value falls outside the tolerable range, the contract *may* be violated. However, it is not certain that a contract is violated when the delivered value is outside the tolerable range. In a networking context, a contract would normally be violated if a QoS parameter value `throughput` falls below the low end, but not when it exceeds the high end. The opposite is true for the QoS parameter value `cost` (in terms of monetary value per bit).

The contract identifier (CID) is used to identify contracts that belong together and whose resources should be treated in unison. A service user suggests which CID should identify this contract when initiating the negotiation phase. The service provider will use the same CID in the resource requests it issues to sustain the contract. When this is applied end-to-end, all the resource reservations that origin from to the same initial request are identified with the same CID. This allows the CID to be used as a guide for handling of resource grants and revocations in order to obtain the effects stated in what is the current resource management policy.

The modules holding contracts that are identified with the same CID, form a subgraph in a resource management graph that connects service users and providers in the system, through all abstraction levels. The modules are the vertices of the graph while the relation "have QoS contract with" forms the edges. Each subgraph is directed, and has an application as root. QoS contracts allow resources belonging to each subgraph to be treated in unison, and the resource management policies decide what the treatment will be.

For instance, assume that the network service has several contracts, and that two of them have the same CID, one with an application and one with the file service. The contract with the file service is a consequence of a contract (with the same CID) between the application and the file service, and is necessary for the file service to carry out its service for the application. If network QoS suddenly drops so that the network service is forced to violate some of its contracts, it may group contracts by CID according to which subgraph they belong to, and only violate contracts within the same subgraph until the resource situation eventually forces it to choose another subgraph as well. This policy will revoke resources from one application at a time, such that as few applications as possible are affected. A priority associated with each subgraph may be applied when choosing the order in which subgraph contracts will be violated.

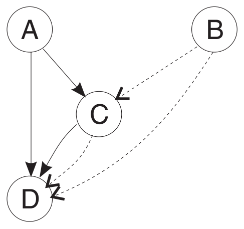

Figure 1: Resource subgraphs

Figure 1 shows that low level service D has four contracts, two of which origin at application A and two at B. The contracts are internally identified with CID_A and CID_B respectively. Those that origin at A are shown with solid lines and those from B are shown with dotted lines. If D is forced to violate some contracts, the architecture ensures that violating contracts tagged with CID_A by itself will not affect contracts stemming from B or B itself.

The monitoring method may either be `poll` or `notify`, and gives the invoker the possibility to specify how the internal information from the invokee is to be conveyed. The `poll` option gives the invoker the responsibility to poll the invokee at points in time appropriate for it, while the `notify` option gives the invokee the responsibility to notify the invoker upon contract violation. Since the invoker may decide which monitoring method it prefers, it may also change methods as part of an adaptation action (including re-negotiation of the contract), if that would suit a new situation better. This will give a user the possibility to choose between adaptivity schemes that favor stability over agility, or vice versa.

The service user may request re-negotiation of a contract at any point in time. The service provider only has that possibility if `notify` is the current monitoring method. Otherwise, re-negotiation that is wanted by the invokee may occur after the next poll by the invoker.

Modules, when acting as service providers, are expected to support both the ability to notify and to handle poll requests. As service users, they are not required to use the management operations of service providers, which enables possibilities for best effort service.

Since monitoring information from the invokee is based on the QoS contracts, it is limited to the parameters included in the contract. It is the invoker's privilege to decide which parameters are to be included in the contract, and thereby limiting the scope of feedback to the parameters it finds important.

3 Integrated QoS-based management

Our architecture consists of modules, where QoS management is integrated into every software module. Each modules is responsible for the collection of the QoS management information it requires. In the design of a module, it is important to emphasize the resources it needs from other modules, how to express those needs, and how to adapt to what it actually gets. Considerations about what service level a module can provide to other modules under varying conditions is also important. The design of software modules would therefore focus on cooperation and adaptation issues rather than performance. A module is not required to use the management operations, and may leave the QoS parameters void during invocation in order to use best effort service.

All software modules may be considered as both service users and service providers.[2] At the lowest level, hardware, device drivers or other low level system software, will detect changes in the form of low level basic events. Their service users will be notified about relevant events—as determined by their contracts, and will map them to higher level events. Upon receipt of a notification, the service user takes the responsibility for internal re-configuration, re-negotiation of the contract, or other control actions. The service user will then determine its own new quality of service level, and notify any of its service users with whom it now holds a violated contract, see figure 2.

The architecture provides no central management body (QoS manager external to the modules) with system wide responsibility for QoS management, for several reasons. Firstly, that does not comply with our notion of integrated QoS management. Secondly, the "end-to-end argument" [Saltzer84] implies that QoS management should be done on the highest possible abstraction level to utilize the highest level of semantic

[2]Except the lowest and highest levels of the software hierarchy.

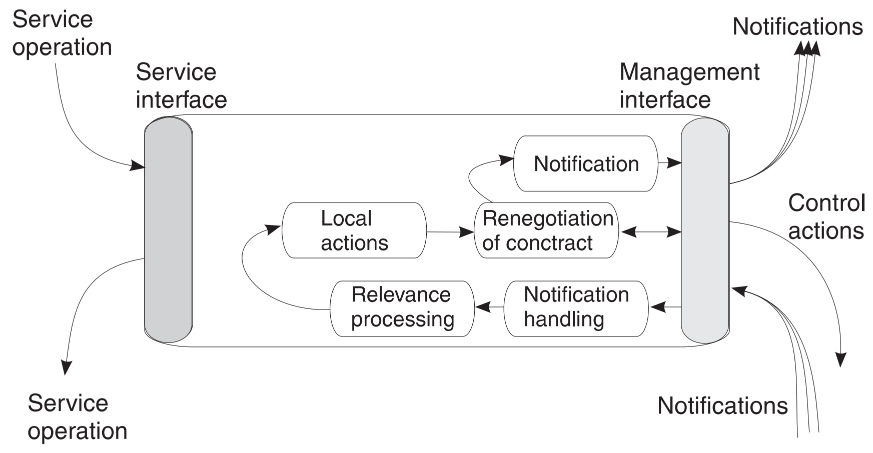

Figure 2: Management processing in every module

information. This level is at the operation invoker (or the user) and not in a separate QoS manager outside the modules. Thirdly, a QoS manager becomes a separate entity that *must* be kept informed about contracts, events and states, and will therefore add complexity to the architecture and make it harder to implement.

This does not preclude the provision of a QoS *helper* that is a repository for management information and that makes the provision of QoS management information and execution of actions more efficient. The responsibility for QoS management is still with the modules that make up the system.

4 The software architecture

The software system of a personal companion will contain a number of resident applications, a number of "volatile" applications that are down loaded from the environment, system support software (including "middleware"), and a multi-tasking operating system that supports processes and threads.

An overview of the structure of the architecture is given in figure 3. Layers are used to describe grouping of functions or services on a similar level of abstraction. The term should not be interpreted too strict since our architecture leans itself towards cooperation between modules rather than towards strict hierarchies.

The middelware layer contains libraries that underpin specific high level abstractions and policies that are shared between a number of applications and therefore not replicated in each application. These abstractions and policies are however considered too specialized to be system services.

The application layer contains the ordinary applications. Each application provides a functional interface and (optionally) a management interface for QoS information and control pertinent to the application. The QM-UI is a special application that implements the user interface to the user controllable QoS management that is not done directly with each application. It provides a system-wide view over QoS management information as well as provide system-wide control commands, and enables user control commands.

The *QH* is a special entity that spans all software layers and contains QoS man-

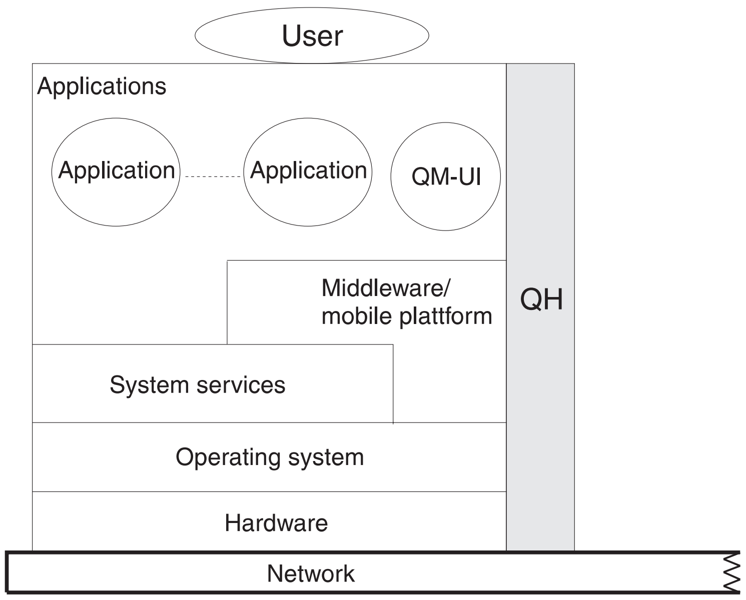

Figure 3: Overview of MobyDick software architecture

agement information and resource utilization information, based on information made available by the modules in the system. Its task is to collect QoS management information across software layers so that they efficiently can be made available to the user through the QM-UI. Modules will register with the *QH* upon creation so that is has access to all operations in their management interface. Notifications from one module to another are sent to the *QH* as well.

The *QH* is not responsible for QoS management, and all modules including the *QM-UI* could perform their task without it. Its primary task is to offer services and abstractions directly to the *QM-UI*, and might be considered a performance optimization. It could be situated in the middleware layer, but its special status with system wide access makes it not properly placed at any level.

We have presented our architecture for managing adaptability in mobile systems. The architecture is general in that is embraces all kinds of resources, and captures adaptability both stemming from the hosting environment and the user. In the following we will outline adaptable security and show an example of how it can be handled in the architecture.

5 Adaptable security

In most settings one will strive for the best security possible. Denial of service due to lack of security resources (certificates or credentials) is normal. However, a truly personal machine is trusted, and this resource is valuable indeed, in fact, a *personal* machine excels over other tools in its resoursfullness: trust. The personal companion can be made such that it can be trusted by its owner, and in a distributed system this is a valuable resource indeed.

Due to short or long term lack of resources, some secure communication may only be completed after a more secure but more costly security scheme is traded for a less costly and less secure scheme. The gain is less risk of denial of service. Cost is usually measured in messages that has to be transmitted. The fundamental problem in adaptable security is that trust can not be quantified, and thus not treated algorithmically. In other words, changes of security policy cannot be done by the software systems alone. Only the user, who is responsible for "trust management", should take decisions based on his feeling of trustworthiness and knowledge of the environment in which the personal companion is running at the moment.

There is a path of QoS mappings that crosses several abstraction levels between resources and security issues. Our architecture allows dynamic security issues to be expressed and treated in the same way as other qualitative issues.

5.1 Security requirements for a personal companion

If the personal companion is to replace items like cash, cheque book, credit cards, passport, keys, diary, etc., it must be able to communicate securely with foreign services. What constitutes security (privacy or integrity, or both) will depend on the particular service. That is, *the user* evaluates the environment, the trust relations and the actual service, and then decides on the "level" of security. Coupled with a resource-poor machine, we obtain an unique environment where security must be adaptable in the same way as the other user oriented services that the personal companion provides.

Since the personal companion will be trusted by its owner, the user will store private data (secrets) in it. This makes it valuable in a very personal way.

In other words, two forces pull in opposite directions at small personal machines. On one hand, their nature implies interaction with unknown and untrusted service providers. On the other, the "closeness" to the user makes them very vulnerable. This is the crux of the security problem.

5.2 Security and QoS

Engagement in secure communication requires assumptions about requirements for encryption, secrecy of keys, and so on. In a system that is designed for a well known and stable mix of principals, the assumptions can be evaluated once to determine whether they are reasonable. In systems where personal companions are used, this approach can not be used. For example, in the office, communication with infrared in probably not problematic since eavesdroppers must be in "line of sight". However, infrared or other broadcast networking technologies are not at all tolerable in public areas, at least without encryption.

Systems employing strong security generally do not permit the user to *lower* the security. Less security might seem undesirable, but in a situation with lack of resources, like no connectivity, there might be a choice between low security and denial of service, which may be totally unacceptable. In general, secure systems pay little attention to the way resources affect security, and vice versa.

Another scenario is one where a clear-text session is underway, and the user is asked to supply a piece of sensitive information. In order to reply, authentication and exchange of a session key must take place, which requires resources. Flexibility is crucial for smooth operation.

We use QoS as the method to manage *changes* in a system and thus how, and when, to change from one strategy to another. Whatever policy is changed to or from, the available alternatives must be solid and argued for based on their quality. The different policies by themselves has nothing to do with QoS.

Lowering the level of security is not done by changing to a shorter encryption key. Such algorithmic “tricks” have only marginal effects on the overall power consumption and communication. To us, the largest effect can be seen by exploiting trust, and the user’s knowledge. At best, software can verify that a signature is created with some key, or that a certificate has not expired. Users, on the other hand, must answer difficult questions, such as “Do you believe that this key belongs to the person with this name.” The gap between a key and a person must be bridged by trust. By making this bridge longer, the user can significantly lower the effort needed by the personal companion; this is how the notion of QoS is used.

Security not only “uses” resources in the form of computation, storage and communication bandwidth, it also provide services to other parts of the system. Examples of such a service is authentication. This is why we are convinced that a successful design of a personal companion must enable not only communication services, but security as well to be adaptable, and hence be included in the notion of QoS. And, thus, security must be designed with adaptability in mind.

5.3 An Example

By means of an example, we will argue that adaptable security is not only useful, but also feasible. Assume that a user, by means of his personal companion, is in the process of extracting money from an ATM (Automatic Teller Machine). The machine wants proper authentication of the human, while the human wants all communication to be encrypted in order to hide sensitive information such as account numbers. Any non-trivial authentication scheme can be used [Liebl93, Lampson92]. Here, we will not dwell with the details of any particular protocol, but instead remark that a large number of protocols exists precisely because there are a large variety of possible assumptions that can be made. Here, we assume the protocol in use is a simple one, as we will now describe. After an initial exchange of messages where the identity of each party is claimed, the personal companion receives a certificate stating that a particular public key is controlled by the ATM. The certificate is signed with the bank’s public key, and the validity is limited (by an explicit expire time). Furthermore, the certificate requires on-line verification to make revocation possible

All actions and messages from this point on, is to strengthen the user’s confidence in the ATM’s public key. First, by fetching the bank’s public key, the user can verify that the certificate was indeed correctly signed. Then, the user fetches and verifies, by means of the bank’s key, the public key of the on-line verification service. A new exchange of messages follows, after which the user can be confident that the ATM’s key is still valid. At last, the exchange of nounces can take place to ensure that the ATM is present now [Burrows90].

The actual number of messages can be high, depending on the actual protocols and how the infrastructure is organized (how is naming implemented, for example). However, the user might also use other means to verify that the received key is indeed controlled by the ATM. If he is inside one of the bank’s branches, chances are high that the ATM is indeed owned and operated by the bank. In this case, the user might trust his own judgment, and skip all the messages required to verify the certificates. This way, the user can act on notifications from the machine on low battery status, for example.

Withdrawing (real) money from an ATM is, of cource, an extreme example. However, in general, it demonstrates that the user is able to adapt the security "on the fly" by changing what is trusted. Such an ability is precisely what is needed to include security in a personal companion where QoS is used to convey changes is the system itself or the environment.

The number of assertions that can be trusted, and thus are interesting in an adaptive system, includes time, public keys, whether a principal is capable of generating good keys, whether communication is private, and so on. All of these can change, and give rise to a different communication pattern (and thus energy consumption). To sum up, security is a well suited service to manage by means of QoS mappings.

6 Conclusion

We have outlined our notion of Quality of service and its use in the structuring of adaptive software systems for mobile computers. Rather than emphasizing on providing QoS guarantees, i.e. how to schedule the available (stable) resources to meet the total amount of demands, our notion of QoS is used to convey relevant and timely management information between service users and providers on the correct abstraction level. Moreover, it captures adaptability both stemming from the hosting environment and user commands.

As an example of how to employ QoS we have chosen to focus on security. We have shown how to use trust to adopt security to changes stemming from the environment.

In the scope of the MobyDick and GDD projects, full QoS architectures covering diverse areas like power management and consistency, are being worked out. Selected test applications with necessary system services, conforming to the QoS architecture is being developed. Key parts of the system support, like seamless switching between network technologies [Brattli96], is already implemented.

Acknowledgments

The collaboration with the members of the MobyDick and GDD projects, in Pisa, Twente, and Tromsø, has been invaluable for this work. We thank you all.

References

[Brattli96] Dag Brattli. *The Software Network, Providing Continous Network Connectivity for Multihoming Mobile Hosts*. PhD thesis. December 1996.

[Burrows90] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, **8**(1):18–36, February 1990. Also available in the Proceeding of the 12th SOSP., Litchfield Park, AZ, USA, 3–6 Dec. 1989. Published as *ACM Operating System Review*, Vol. 23, No. 5, pp. 1–13, December 1989. A fuller version was published as DEC System Research Senter Report No. 39, Palo Alto, California, USA, February 1989. Also presented in *Proceedings of the Royal Society of London*, Series A, 426:233–271, 1989.

[Davies93] N. Davies, G. Coulson, and G. S. Blair. Supporting Quality of Service in Heterogenous Networks: From ATM to GSM. Internal Report MPG-93-26. Department of Computing, Lancaster University, Bailrigg, Lancaster, Nov. 1993.

[Forman94] G. H. Forman and J. Zahorjan. The challenges of mobile computing. *IEEE Computer*, **27**(4):38–47, Apr. 1994.

[Huard96] J.-F. Huard, I. Inoue, A. A. Lazar, and H. Yamanaka. Meeting QOS Guarantees by End-to-End QOS Monitoring and Adaptation. *Proceedings of the Fifth IEEE International Symposium On High Performance Distributed Computing (HPDC-5)*, Aug. 1996.

[Imielinski94] T. Imielinski and S. Viswanathan. Adaptive Wireless Information Systems. *Proceedings of the SIGDBS Conference* (Tokyo, Japan), Oct. 1994.

[Kaashoek94] M. F. Kaashoek, T. Pinckney, and J. A. Tauber. Mobisaic: An Information System for a Mobile Wireless Computing Environment. *Proceedings of the 1st IEEE Workshop on Mobile Computing Systems and Applications* (Santa Cruz, CA, USA), Dec. 1994.

[Lampson92] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distribued systems: theory and practice. *ACM Transactions on Computer Systems*, **10**(4):265–310, November 1992. Also available as a technical report from ftp://gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-083.ps.gz. A preliminary version of this paper appeared in the Proceedings of the Thirteenth ACM Sypomsium on Operation Systems Principles.

[Liebl93] Armin Liebl. Authentication in distributed systems: A bibliography. *Operating Systems Review*, **27**(4):31–41. ACM, October 1993.

[Saltzer84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, **2**(4):277-288, Nov. 1984.

[Satyanarayanan94] M. Satyanarayanan, B. Noble, Puneet Kumar, and Morgan Price. Application-Aware Adaptation for Mobile Computing. *6th ACM SIGOPS European Workshop* (Dagstuhl Castle, Wadern, Germany, Sept. 1994), pages 1–4, Sep. 1994.

[Schilit94] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. *Proceedings of the 1st IEEE Workshop on Mobile Computing Systems and Applications* (Santa Cruz, CA, USA), Dec. 1994.

[Weiser91] Mark Weiser. The computer for the twenty-first century. *Scientific American*, **265**(3):94–104, Sept. 1991.