Stublets: A Notion for Mobility-Aware Application Adaption*

Dietmar Kottmann, Christian Sommer
University of Karlsruhe, Institute of Telematics
D–76128 Karlsruhe, Germany
{kottmann|sommer}@telematik.informatik.uni-karlsruhe.de

1 Introduction

Today's wireless technologies have some characteristics that are anachronistic in the context of current fixed networks. Unpredictable quality of service fluctuations and high cost are the challenges of a mobile computing environment that an application has to cope with (cf. [BCK+95]). Generally, a client-server-style application designed for untethered devices can choose between three interaction paradigms to address these shortcomings: replication of data, remote operation execution, and delegation of functionality to the fixed net (cf. [CGH+95]). In this position paper we propose a model whose aim is to reduce the usage cost of wireless links by adaptively employing these techniques.

2 The Stublet Approach

Any application normally comprises components dedicated to local and remote purposes. Whereas presentation facilities (e.g. graphical user interfaces) are of purely local relevance, some operations of the application domain will access remote data and services shared with other applications. As these operations potentially cause communication traffic across high cost wireless links, we propose to group them together into special modules and separate them from the rest of the application. From now on, we restrict our scope to these units, which we call *stublets*. These stublets can be executed using replication, remote operations, or delegation. If all the objects required by a stublet are locally available (i.e. replicated), we can access them directly even in case of disconnection. Otherwise, it is possible to use conventional remote operations, or to ship the whole stublet as a script to the fixed net where it is executed on behalf of the mobile device. At run-time, a decision function dynamically determines which technique seems to be best. This decision is based on QoS and cost information exposed from the mobile net and semantic knowledge added to the application by the programmer. The overall stublet architecture is depicted in figure 1. From the programmer's point of view, the stublet approach necessitates a modification of the application development cycle (cf. fig. 2).

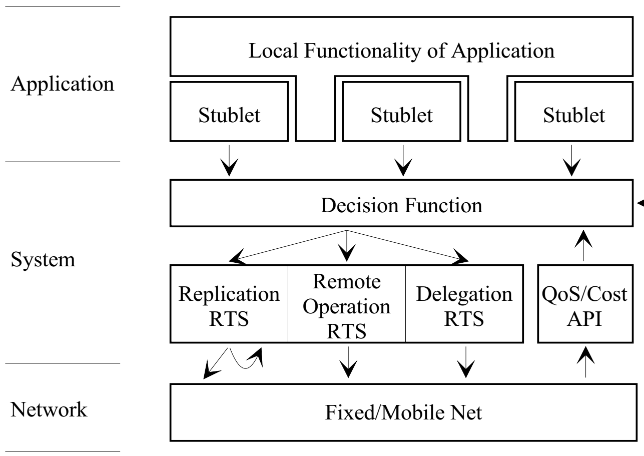

Figure 1: System Architecture with Stublets

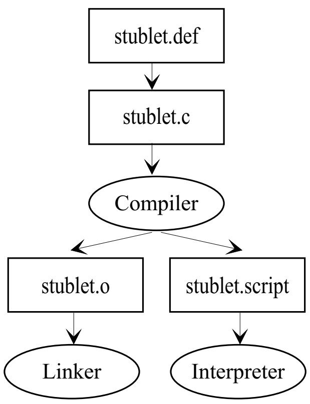

Figure 2: Application Development Process

3 Enabling Technologies

In this section we report about our experience with enabling technologies which we plan to exploit for designing and implementing the stublet approach.

*This work was partly supported by the German Research Council (Deutsche Forschungsgemeinschaft DFG) under grant SFB 346-A6 and grant GRK 209/2-96.

3.1 Object Replication in MISTRAL

Below we sketch our previous work on semantic-based replication for disconnected operations. As we cannot discuss the *MISTRAL* system exhaustively, we refer the reader to [Kot96] for further details. Our approach draws from the repeatedly reported fact (cf. [SKM+93]) that mobile users know in advance what objects they need and how they want to use them. Therefore, MISTRAL bases its decisions on the knowledge of the operations a user intends to execute disconnectedly and on semantic relationships between them expressed via an *invalidation* and a *dependency* relation. Moreover, MISTRAL offers three basic functions: *disconnection*, *actualization*, and *reintegration*. On disconnection it is verified whether pessimistic guarantees for the requested operations can be granted. Exploiting the semantics, it is even possible to let multiple users execute certain modifying operations (e.g. when they commute) disconnectedly. On actualization the states of two replicas are mutually updated. On reintegration one replica is melted with another, so that it is consistent with all operations executed on both replicas. MISTRAL allows users to execute operations optimistically, too. Anomalies which inevitably can result from optimistic operations are only visible to the user who risked the optimistic mode. However, anybody who sticks to pessimistic guarantees never has to fear that one of his operations will become invalid afterwards.

We have implemented MISTRAL as C++-wrappers that enable the communication with the run-time part and perform necessary logging. Figure 3 shows the performance for reintegration of pessimistically performed disconnected updates, measured on two DEC AXP 3000/500 connected via a lightly loaded Ethernet.

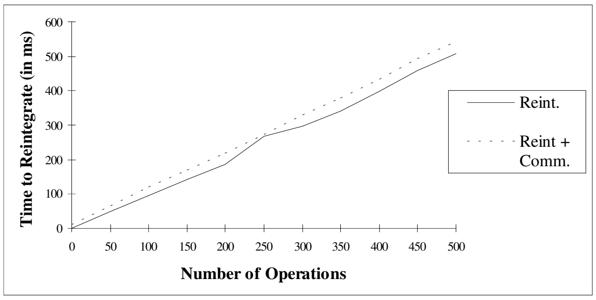

Figure 3: Reintegration Performance

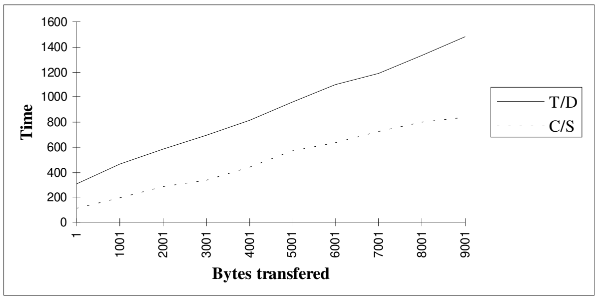

Figure 4: Delegation vs. Remote Call

3.2 Delegation with Trustee/Delegate

We have built the *Trustee/Delegate*-System to evaluate the prospects of delegation. Details can be found in [KWP+96]. It consists of three components: (1) a trustee, which is part of the client and allows deferred synchronization of delegated operations as well as integrated callback processing, (2) a proxy delegate, which lets scripts mimic original RPC-calls to ONC-servers, and (3) a development environment, which compiles complex operations defined in the **D**elegate **I**nstruction **L**anguage (DIL) into a script interpretable by the proxy delegate. The most important aspect for this paper is to give an impression of the run-time overhead which is imposed by delegation due to shipping the script, mimicking RPC-protocols and interpreting the delegate code. This can be seen in figure 4 which compares a conventional RPC-call from a mobile client to a RPC via shipping a script. This figure has been obtained in the mobile computing lab of Daimler Benz.

4 Policies

In this section we exemplarily investigate possible policies for dynamically choosing between replication, remote operations, and delegation.

4.1 When to Trigger Policies

There is a natural trigger for the decision between remote operations and delegation. The superiority of one of these two approaches varies with the QoS provided and cost charged by the currently available network. As a consequence, the predominance of one is just temporary, so that the choice should be made each time a stublet is invoked. On the other hand, replication seldomly performs better when only single invocations are regarded. Rather, it creates a long-term capability to work locally instead of being forced to use expensive untethered links. The cheaper the price for creating this capability, the higher the probability that its benefits outweigh its costs. Hence, we should decide whether to replicate objects or not at a point at which the cost of communication is up to change. For simplicity we only tell wired networks, which are free of charge, from high cost wireless ones. Under this premise, the decision for or against replication has to be triggered exactly at this border (cf. fig. 5).

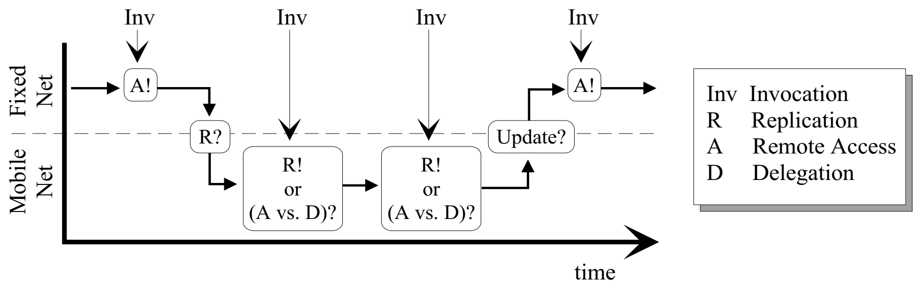

Figure 5: Triggering Policies

4.2 Replicated vs. Remote Data

MISTRAL needs semantic information about the objects to replicate, which is reflected by several extensions to the interface definition of an object (cf. fig. 6). First of all, objects are classified in *replicatable* and *non-replicatable* ones. Replication-resistant objects either depend on some resources of the fixed network (e.g. a printer interface) or provide real-time information (e.g. share exchange rates). In addition, for replicatable objects the *invalidates*-relation defines which operations can be executed independently in disconnected mode. A replication protocol allocating the right to execute sets of operations on replicas can guarantee serializability, if it assures that the invalidates-relation between those sets keeps acyclic. To accomplish this, MISTRAL uses *PastLocks* and *FutureLocks*. Furthermore, the interface includes information necessary to resolve conflicts in case of optimistic operations (*dependencies*, *conflict_types*). There must be semantic knowledge in the stublet definition, too. A mobile user has to specify which operations on which objects he wants to execute pessimistically while disconnected. Moreover, he can group the objects into consistency-groups (the so-called *timezones*) to increase the number of pessimistic guarantees. The annotations to the object and stublet interfaces are shown in figure 7 for a stocktaking scenario.

```
interface Stock replicatable deterministic {
   attribute table<string,int> contents;
   int quantityOnStock(string itemId);
   void addToStock(string itemId, int quantity)
      invalidates(quantityOnStock) when(quantity != 0);
   dependencies {...};
   conflict_types {...};
};
```

Figure 6: Interface Definition for Replication

```
stublet void AvailableGoods(Stock a_stock) {
   requires_object a_stock {
      timezone: orderprocessing;
      operations {
         quantityOnStock: pessimistic;
      };
   };
};
```

Figure 7: Stublet Definition for Replication

To sum up, when a mobile host is about to enter the mobile net, it is tested what rights must be kept at the fixed net. Their definition is an administrative task which can be done when business processes and the required objects are defined. Moreover, the rights already allocated by other mobile nodes have also to be taken into account. Then replicas are generated for all timezones whose requested pessimistic operations could be assigned the required rights, i.e. that those rights do not result in a cyclic invalidates-relation.

4.3 Remote Access vs. Delegation

Below we discuss the question when to choose remote operations and when to prefer delegation. In order to reduce the usage cost of a wireless link, communication traffic must be avoided. To simplify the discussion we assume a packet-oriented data network, i.e. the cost is proportional to the amount of data to be transferred. Hence, the decision whether delegation will be worth its overhead mainly depends on the sizes of parameters and intermediate results of remote operations. Consequently, the internal data flow of a stublet has to be assessed in advance. But the type information of interfaces is insufficient for doing so because the exact size of dynamic data structures (e.g. sequences) is unknown. Therefore, we suggest annotating the server interfaces with a *selectivity* statement, which is a comment denoting the sizes of the return results of an operation. Hereby, a stublet programmer is capable of defining the functions *RemCost* and *DelCost* which estimate the cost of remote access and delegation respectively.

In our first scenario a mobile client wishes to perform a conversion from dvi- to ps-format before printing a file remotely. A ps-file has approximately two times the size of the corresponding dvi-file. This knowledge should be indicated by a *static* selectivity statement in the interface of the printing service. Static means that the output size of an operation is a constant or a function of the input. With remote operations the total amount of data transmitted is five times the size of the dvi-file, if we ignore the small contribution of the printer status. With delegation only the initial dvi-file and the *DviPsLpr* script have to be transmitted (cf. fig. 8). Note, that it is necessary to define *DelCost* explicitly because stublets can get back results of unpredictable size.

The next example illustrates the benefits of *dynamic* selectivity, i.e. that the amount of data cannot be predicted and must be determined at run-time. A traveling salesman wants to inquire the availability of a product which is comprised of several components. Thus he has to retrieve the component list from a component database and then to check if each single part is on stock. Using remote operations, we would have to ship the complete component list as well as many inquiries over the net. Using delegation, we only need to transfer the final result besides the script. The superiority of one of these approaches depends on the number of components. As this number is neither known at the time the stublet is defined nor at the time of invocation, we allow the stublet programmer to use remote operations (like *numberOfComp()*) in the definition of the *RemCost* function (cf. fig. 9). It is the responsibility of the server to provide these dynamic selectivity operations.

```
stublet Status DviPsLpr(File DviFile) {
  transmission {
    RemCost = (1+2+2) * DviFile;
    DelCost = DviFile + script;
  };

  File PsFile
       = PRINT_SERVICE->dvips(DviFile);
  return(PRINT_SERVICE->lpr(PsFile));
};
```

Figure 8: Exploiting Static Selectivity

```
stublet boolean Available(int productId) {
  transmission {
    RemCost = COMP_DB->numberOfComp(productId) * (Component+100);
    DelCost = script;
  };
  ComponentList list = COMP_DB->getComponents(productId);
  for (int i=0; i<list.length; i++)
    if (!STOCK->onStock(list[i].id)) return FALSE;
  return TRUE;
};
```

Figure 9: Exploiting Dynamic Selectivity

5 Summary and Discussion

The performance and cost disparity between wireless and wired communication channels motivates the demand for application-level adaption by dynamically choosing between replication, remote access, and delegation. A *stublet* denotes the unit to which one of these techniques is applied. Semantic-based replication at the boundary to the mobile net enables local access to objects without any communication cost at all, after the mobile node has left the fixed net. Nevertheless, due to consistency requirements we cannot generally avoid the need for remote services. In this case, launching a complete stublet as a script may be superior to invoking each operation in the stublet seperately. Transparent *selectivity* statements in the server interfaces are the means by which a stublet programmer can estimate cost functions for the decision process at run-time.

The following problems have not been addressed yet and need further research: Firstly, we have to include more optimization criteria to the decision function, e.g. reducing connection time charges or minimizing the end-to-end delay for a stublet invocation. In this context the currently available quality of service of the mobile net, such as bandwidth and latency, will play an important role. Secondly, it should be examined if it is possible to generate the cost function of a stublet (semi-)automatically. Perhaps data flow analysis inside the compiler can be exploited for this purpose. Thirdly, we must settle the question how the server-side overhead for the provision of dynamic selectivity operations can be kept small. From a client's point of view, executing a remote service is obviously unreasonable just to determine the sizes of the results. Instead, the server should employ a pre-emptive strategy to efficiently provide selectivity information. Finally, we have to develop a prototype for the stublet approach taking advantage of our experiences in implementing MISTRAL and Trustee/Delegate.

References

[BCK+95] Bagrodia R., Chu W.W., Kleinrock L., Popek G.: *Vision, Issues, and Architecture for Nomadic Computing*. IEEE Personal Communications, Vol. 2, No. 6, Dec. 1995, pp. 14–27.

[CGH+95] Chess D., Grosof B., Harrison C., Levine D., Parris C., Tsudik G.: *Itinerant Agents for Mobile Computing*. IEEE Personal Communications, Vol. 2, No. 5, Oct. 1995, pp. 34–49.

[KWP+96] Kottmann D.A., Wittmann R., Posur M.: *Delegating Remote Operation Execution in a Mobile Computing Environment*. To appear in: ACM – Baltzer Mobile Networks and Nomadic Applications.

[Kot96] Kottmann D.A.: *Replication-Support for Advanced Mobile Applications*. To appear in: World Conference on Mobile Communications, Canberra, Australia, Sep. 1996.

[SKM+93] Satyanarayanan M., Kistler J.J., Mummert L.B., Ebling M.R., Kumar P., Lu Q.: *Experiences with Disconnected Operations in a Mobile Computing Environment*. In Proc. USENIX Symp. on Mobile and Location Independent Computing, Cambridge, MA, Aug. 1993, pp. 11–28.