Software Engineering for Scalable Distributed Applications

Maarten van Steen
Vrije Universiteit
steen@cs.vu.nl

Stefan van der Zijden
CAP Gemini
SZijden@inetgate.capgemini.nl

Henk J. Sips
Delft University
sips@cs.tudelft.nl

Abstract

A major problem in the development of distributed applications is that we cannot assume that the environment in which the application is to operate will remain the same. This means that developers must take into account that the application should be easy to adapt. A requirement that is often formulated imprecisely is that an application should be scalable.

We concentrate on scalability as a requirement for distributed applications, what it actually means, and how it can be taken into account during system design and implementation. We present a framework in which scalability requirements can be formulated precisely. In addition, we present an approach by which scalability can be taken into account during application development. Our approach consists of an engineering method for distributing functionality, combined with an object-based implementation framework for applying scaling techniques such as replication and caching.

1 Introduction

In the past 20 years, we have seen the integration of computing and communication facilities into large-scale enterprise-wide networked environments, capable of supporting large numbers of users and applications. Large-scale networking has also introduced interoperability problems. Current environments are highly heterogeneous consisting of different hardware platforms, operating systems, and network architectures. This, in turn, has led to new developments in which common middleware solutions are sought that allow us to integrate different applications. Examples include OLE/DCOM, CORBA, and Java RMI.

These developments are now leading to a new generation of information systems that are inherently distributed across large networks of computers, and capable of interoperating with other systems as in the case of, for example, federated databases. A major problem in the development of these distributed information systems, is that we cannot assume that the environment in which the system is to operate will remain the same over time. This means that developers must take into account that the system should be easy to adapt to meet requirements that are unknown during the development process. At best, such unknown requirements are formulated imprecisely. One particularly fuzzy requirement is that the information system should be **scalable**. Informally, this means that the system should easily accommodate higher performance levels, because, for example, the size or geographical dispersion of the set of users changes. At the same time, systems should be scalable in the sense that they can easily be adapted to cooperate with future applications.

Scalability is the topic of this paper. In particular, we concentrate on scalability as a requirement for distributed information systems, what it actually means, and how it can be taken into account during system design and implementation. We advocate that scalability requirements can and should be formulated precisely.

Our main contribution is that we explicitly and exclusively focus on the role of scalability in the design of distributed applications. To our knowledge, there are very few papers having the same objective. In a sense, this paper supplements a treatise on scalability by Neuman [1]. However, we concentrate on scalability from the perspective of software engineering. In particular, rather than providing general design guidelines, which have also been discussed in a report by Schwartz [2], we discuss the architectural support needed for development of scalable distributed applications and suggest some solutions.

The paper is organized as follows. In Section 2 we provide a formal framework for specifying scalability as a systems requirement. How scalability requirements can be captured during the development process is discussed in Section 3 where a methodology is presented for distributed information design. Scalability support should be provided during the implementation phases as well, in particular by means of a scalable implementation framework. Such a framework is discussed in Section 4. We conclude in Section 5.

2 Defining Scalability

To motivate the following discussion, let us consider a common, informal definition of scalability. An application is

called scalable if it can "accommodate whatever performance level or number of users necessary by simply adding resources to the system [...]. A desirable form of scalability is a resource cost that is at most linear in some measure of performance or usage" [3].

In this definition, a distinction is made between users, performance, and system resources. When taking a closer look at the definition, many questions come to mind. For example, is only the number of users interesting from a scalability point of view, or could there be other issues of interest as well? Furthermore, it does not seem reasonable to require that *any* performance level should be accommodated. Also, whether adding more resources is the only solution to scalability remains to be seen. It is not hard to imagine that certain applications can never scale without adaptations, no matter how much resource capacity is available. Finally, why resource costs should be limited to linear growth is unclear. It may well be the case that in certain cases scalability costs should be limited to sublinear growth, or, on the other hand, are allowed to grow superlinearly.

In this section, we provide a formal framework for reasoning about scalability of distributed applications. We argue that scalability can be formulated precisely, but will, in general, depend strongly on the application.

2.1 Basic Model

We consider a distributed application as a number of cooperative programs running on multiple machines, sharing a collection of data. We separate its clients and execution environment as shown in Figure 1, leading to four surrounding environments:

The **client environment** consists of end users and processes that make use of the services implemented by the application. The client environment itself can be distributed: users and processes possibly reside at different locations while being part of the same application.

The **development environment** consists of application developers that maintain the distributed application, by removing, updating, and adding new parts. It also comprises the methods, tools, and techniques for application development.

The **administration environment** consists of system managers responsible for enabling execution, and monitoring of the application for one or more execution environments. It also comprises the procedures and tools by which the process of administration is organized.

The **execution environment** models the underlying infrastructure in which the application is executed. It contains resources, operating systems, networks, and middleware that the application needs to execute.

In the model presented here, growth of the client environment is the source for scaling an application; the execution, administration, and development environment should

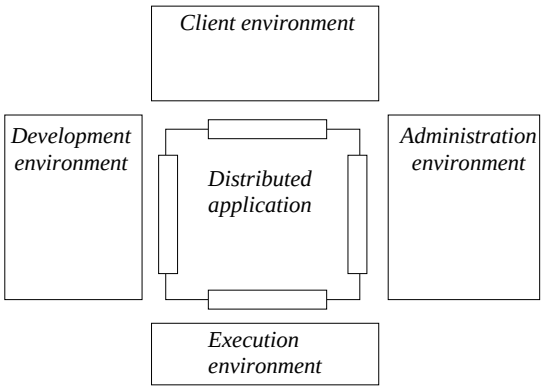

Figure 1: A model for defining and reasoning about scalability.

offer support that allows the application to scale. An alternative approach is to also take into account scalability requirements from the perspective of the administration and development domains. However, such alternatives are not further pursued in this paper.

To formally define scalability, we associate a number of **attributes** $Attr_1, \ldots, Attr_N$ with the client environment. Examples of such attributes are the number of clients in the environment, the number of requests issued by the clients, the volume of data passed between the application and clients, the size of the geographical region in which clients reside, and the geographical distribution of clients within a specific region.

An instance of the client environment is represented by a vector $\mathbf{a} = \langle a_1, \ldots, a_N \rangle$ of attribute values, with $\mathbf{a}[i] = a_i$. We assume that the values of an attribute are totally ordered.

For example, suppose we are interested in the scalability of an application only with respect to (1) the number of end users and (2) the size of the region where those users reside. We can then represent the client environment by a vector of length two. We may decide to count users in units of 10 and to express the size of the region as either SMALL, MEDIUM or LARGE, so that a specific environment can be represented as the vector $\langle 40, \text{MEDIUM} \rangle$. Note that laying down scalability requirements implies that we first explicitly have to choose relevant attributes and their respective range of possible values.

Furthermore, we associate a **performance measure** $Perf_A$ with the distributed application A. Performance is expressed in some numerical **performance measure unit**. A low value of the performance measure indicates poor performance. An example is the number of services that are completed per time unit, also called throughput. We assume that attributes and performance measure are chosen such that an increase of an attribute value decreases performance.

Analogous to the attributes of the client environment, we model the execution environment as a collection of re-

sources $Res_1, \ldots, Res_M$. An instance of the execution environment, which represents the available resource capacity, is expressed as a vector $\mathbf{r} = \langle r_1, \ldots, r_M \rangle$. We assume the performance of an application also depends on the available resource capacity, and in particular, that an increase of resource capacity leads to better performance.

We can thus express the performance of an application A as a function $Perf_A(\mathbf{a}, \mathbf{r})$ that is decreasing in $\mathbf{a}$ (i.e., $\frac{\Delta Perf_A}{\Delta \mathbf{a}} \leq 0$) and increasing in $\mathbf{r}$ (i.e., $\frac{\Delta Perf_A}{\Delta \mathbf{r}} \geq 0$).

The **cost** of available resource capacity $\mathbf{r}$ is denoted $Cost(\mathbf{r})$. Costs are expressed in some **cost unit**. The cost for realizing performance $Perf_A(\mathbf{a}, \mathbf{r})$ is denoted $Cost_A(\mathbf{a}, \mathbf{r})$. Finally, $\mathbf{r}_{\min}(\mathbf{a})$ denotes the resource capacity needed to realize $Perf_A(\mathbf{a}, \mathbf{r})$ at minimal costs, that is, if $Perf_A(\mathbf{a}, \mathbf{r}) = Perf_A(\mathbf{a}, \mathbf{r}_{\min}(\mathbf{a}))$ then $Cost_A(\mathbf{a}, \mathbf{r}) \geq Cost_A(\mathbf{a}, \mathbf{r}_{\min}(\mathbf{a}))$.

2.2 A Formal Definition of Scalability

Now consider a distributed application A that has been initially designed for a client set $\mathbf{a}_{\text{ref}}$, that is, $\mathbf{a}_{\text{ref}}$ acts as a reference point for possible adaptations. Associated with $\mathbf{a}_{\text{ref}}$ is an assumed available resource capacity $\mathbf{r}_{\text{ref}}$ at minimal costs, that is, $\mathbf{r}_{\text{ref}} = \mathbf{r}_{\min}(\mathbf{a}_{\text{ref}})$.

Denote by $\mathbf{a}_{\text{ref}}\langle i : a \rangle$ the vector of attribute values which has the same values for its elements as $\mathbf{a}_{\text{ref}}$, except for the i^{th} component, which has the value a. We can now define scalability as follows.

Definition. Let $\gamma(\mathbf{a})$ be a function in attribute values $\mathbf{a}$ returning values in the same cost unit as $Cost_A$. Furthermore, let $\delta(\mathbf{a})$ be a function in $\mathbf{a}$ returning values in the same performance measure unit as $Perf_A$ with $\forall \mathbf{a} :: \delta(\mathbf{a}) \geq 0$ and $\delta(\mathbf{a}_{\text{ref}}) = 0$. An application A is scalable in attribute $Attr_i$ for values up to a maximum $a_{\max}$ with respect to γ and δ if the following properties hold:

P1: A can accommodate values $\mathbf{a}_{\text{ref}}[i] < a \leq a_{\max}$.

P2: $\forall \mathbf{a}_{\text{ref}}[i] < a \leq a_{\max} \exists \mathbf{r} ::$
$Perf_A(\mathbf{a}_{\text{ref}}, \mathbf{r}_{\text{ref}}) - Perf_A(\mathbf{a}_{\text{ref}}\langle i : a \rangle, \mathbf{r}) \leq \delta(\mathbf{a}_{\text{ref}}\langle i : a \rangle)$

P3: $\forall \mathbf{a}_{\text{ref}}[i] \leq a \leq a_{\max} ::$
$Cost_A(\mathbf{a}_{\text{ref}}\langle i : a \rangle, \mathbf{r}_{\min}(\mathbf{a}_{\text{ref}}\langle i : a \rangle)) \leq \gamma(\mathbf{a}_{\text{ref}}\langle i : a \rangle)$

The first property states that the application can actually accommodate a growth of attribute value $\mathbf{a}_{\text{ref}}[i]$. For example, let $Attr_i$ denote the number of users of an application. If the maximum number of users has been hard coded to 100, then, by definition, the application's scalability is limited to that number. So, setting $a_{\max} > 100$ *in the scalability requirements* automatically disqualifies the application as being scalable according to those requirements.

Underlying property **P2** is our assumption that increasing the value of attribute $Attr_i$ from $\mathbf{a}_{\text{ref}}[i]$ to a leads to a performance degradation. The second property states that we can increase the available resource capacity in such a way that this degradation can be compensated, although some degradation may be acceptable. Degradation tolerance is expressed by means of the **performance degradation bound** δ.

For example, if we take $\delta(\mathbf{a}) \equiv 0$ we are demanding that performance degradation can be *fully* compensated by (only) adding more resources. However, this may be too rigid in some cases. Suppose we have a client/server application initially designed to handle at most 40 clients without noticeable performance degradation. With $\delta(\mathbf{a}) \equiv 0$ we would be forced to add an extra server as soon as the number of clients becomes 41. However, by accepting some performance degradation, expressed by specifying that $\delta(\mathbf{a}) \equiv K$ for some $K > 0$, we could allow up to, say, 50 clients before adding an extra server to meet our scalability requirements.

The third property (**P3**) limits the costs that are allowed to compensate performance degradation. In particular, we require that there is a **reference cost bound** γ that limits the maximal costs for increasing resource capacity. Note that the costs can be expressed any way that is felt appropriate. We could take the actual monetary costs, but also, for example, costs expressed in terms of storage or processing capacity.

Defining scalability in terms of the cost of resource capacity is not always practical as these costs are often hard to measure. This is particularly the case when the execution environment itself offers interfaces that completely hide the resources it offers, as is the case with many middleware solutions. In these cases, scalability can be formulated only in terms of performance degradation. This means that property **P3** should be ignored, at that scalability requirements are formulated entirely in terms of the performance degradation bound δ.

2.3 Discussion

An important issue in our framework is the distinction between *extensibility* and *performance*. Often, scalability is formulated only in terms of performance. In our framework, we explicitly account for the fact that an application should be able to accommodate certain ranges of attribute values, like being able to administrate a large number of users. Extensibility in this sense, is independent from performance, and refers solely to being capable of coping with certain attribute values in the administration environment. Note that the developer, or the customer for whom the application is being built, is responsible for defining the set of relevant attributes.

Another important aspect of both definitions are the additional bounds that limit the increase of costs, and the degradation of performance, respectively. It is up to the customer to define these bounds. For example, it may be enough to state that costs for scaling the application by adding resources should be limited to a linear increase. On the other hand, it may also be the case that costs should remain the

same, but that the application should be able to support twice as many users at half the performance.

The point to note is that scalability requirements can be formulated *precisely* in our framework. The minimum needed are definitions of the attributes of the client environment, a simple performance measure for the application, and the performance degradation bound δ. If resources are taken into account, we need the resource cost measure as well as the reference cost bound γ. Defining these components for general applicability does not make much sense as they strongly depend on the usage and nature of an application.

In this sense, our approach is more liberal than the few alternative attempts at formally defining scalability. To our knowledge, scalability has been formally defined only in the field of high-performance computing. In a recent study, Kuck provides precise definitions of performance scalability [4]. Using our terminology, the client environment in Kuck's model consists of a single client. The only attribute that is taken into account is the (data) size of the problem that the application needs to solve. The execution environment consists of a high-performance machine, for which only the processors are considered as resources that affect scalability. The performance measure is the speed-up that can be accomplished by increasing the number of processors. Speed-up is the *defacto* performance measure in high-performance computing.

Kuck explicitly gives several values for the reference cost bound γ. For example, he states that high-performance is reached only if speed-up exceeds $P/2$, where P is the number of processors. An application is then high performance data-size scalable only for those sizes of the input set for which the speed-up exceeds $P/2$. Similar bounds are used for defining other classes of scalability. The drawback of Kuck's definitions is that they deal only with data size and number of processors, and are difficult to apply to areas other than parallel computing. Moreover, Kuck hardly motivates his scalability bounds. This seriously limits their applicability to other cases.

3 Engineering Methods: AD-DIS

So far, we have merely provided a framework for specifying scalability requirements. We have not said anything on how we can actually build scalable distributed applications. We distinguish two issues. First, supplementing existing methods and tools for developing applications, we propose a systematic approach toward specifying the distribution of functionality of an application. Second, we propose to use a scalable implementation framework that assists a developer in implementing distribution policies. In this section, we concentrate on a method for specifying distribution of functionality. Implementation is discussed in Section 4.

3.1 Distributed-Software Engineering

It has been recognized that the distribution of functions and data severely affect the overall quality of an information system and that distribution should be taken into account early in the development process [5]. Unfortunately, well-established development methods mostly ignore distribution aspects and at best consider them during phases of technical design [6]. A solution to this lack of support is to use a supplemental development method that deals solely and explicitly with distribution aspects. Architecture Design for Distributed Information Systems (AD-DIS) from Cap Gemini is such a method [7].

The goal of AD-DIS is to find an optimal location for the functional components of an application. The functional and quality requirements identified during a previous requirements phase are taken as a starting point for locating components, along with the capabilities and limitations of the execution environment.

When designing an architecture for distributed applications, we have to face a number of hard problems. For example, there is often much design information to consider. Also, we generally have to face conflicting requirements, such as demands for high performance while meeting severe security constraints. Moreover, distribution can take place with respect to different issues. For example, we can distribute functions for optimizing manageability, security, performance, etc.

The process of architecture design can easily become uncontrollable and lengthy, so that justifying the final result may eventually turn out to be difficult. To alleviate problems, AD-DIS uses *multiple abstraction levels* and *scenarios*.

Multiple abstraction levels are deployed to handle the large volume of design information. The following three levels are distinguished in AD-DIS:

On the **conceptual level** all relevant design information is collected and modeled from a topographical point of view. Important issues concern quality requirements such as availability, performance, and actuality of data. Also, topographical issues are taken into account such as where specific functions are needed, or where they should never be made available, etc. The result is a *conceptual distribution design*.

On the **logical level**, developers decompose functions into physically distributed units. The result is a set of nondistributable components, called a *logical distribution design*. The data and functions encapsulated by a component will always be mapped to the same location, although replication is still possible. Based on this decomposition, a number of distribution schemes are worked out, but discarding the capabilities and limitations of the execution environment. In other words, ideal distribution alternatives are constructed, analyzed, and compared.

Finally, on the **physical level**, the logical distribution design is mapped onto an execution environment, taking capabilities and limitations of the latter into account. Typical decisions that are made at this level are whether or not to replicate a component, or where to physically locate components. The result is coined a *physical distribution design*.

In addition, AD-DIS uses different scenarios at each level to perform *what-if* analyses. Within a scenario, a developer concentrates on optimizing distribution for the sake of only one or two requirements. For example, the logical design can be optimized for performance or availability. Within another scenario, a developer would concentrate on manageability of the distributed information system. By comparing several scenarios, we obtain better insight concerning the effects of distributing the components identified on the logical level. Eventually, a compromise may have to be sought.

3.2 Designing for Scalability

The AD-DIS method can be used to take scalability into account in application development. Consider the situation in Figure 2. A distributed application has a cost-based scalability requirement as illustrated in the graph. Two points are chosen as bounds for the scaling bandwidth: $\mathbf{a}_{\text{ref}}$ and $\mathbf{a}_{\text{max}}$. For both points a distribution design will be made using AD-DIS.

Based on the client environment as characterized by $\mathbf{a}_{\text{ref}}$, a conceptual distribution design $\mathsf{C}(\mathbf{a}_{\text{ref}})$ is made. Taking this design as starting point, different scenarios are considered to derive one or more logical designs. Examples of such scenarios are:

Scenario 1: A scenario in which the components of the application are distributed in such a way that performance is optimized. In our example, the result is the logical distribution design $\mathsf{L1}(\mathbf{a}_{\text{ref}})$.
Scenario 2: A scenario in which the components are distributed for optimal security, resulting in the logical distribution design $\mathsf{L2}(\mathbf{a}_{\text{ref}})$.

The choices made on the logical level deal only with the locality of data and functional components of the application. A typical decision on this level is to choose for central or distributed data management. In our example, the results of the two scenarios are compared for compliance with *all* requirements. One of the two logical designs is chosen as the basis for the physical distribution designs.

We take logical design $\mathsf{L2}(\mathbf{a}_{\text{ref}})$ as the basis for developing a number of physical distribution designs. Again, several scenarios are examined. The possible scenarios depend on the available mechanisms in the execution environment. For example, to implement central data management, three scenarios could be investigated:

Scenario 2.1: access to a physical central database, resulting in physical distribution design $\mathsf{P21}(\mathbf{a}_{\text{ref}})$
Scenario 2.2: a central database with local caching, leading to $\mathsf{P22}(\mathbf{a}_{\text{ref}})$
Scenario 2.3: a scheme with master-slave replication to allow local database copies, leading to $\mathsf{P23}(\mathbf{a}_{\text{ref}})$

The chosen physical distribution design prescribes (1) where components are to be located, and (2) which mechanisms are to be used for distribution support. Note that the actual development of the components can be supported by a separate, perhaps traditional method that may now ignore distribution issues.

To account for scalability requirements, we follow exactly the same development process, but now taking the client environment characterized by $\mathbf{a}_{\text{max}}$ as the starting point. In our example, this eventually leads to the physical distribution design $\mathsf{P22}(\mathbf{a}_{\text{max}})$ which is different from the design for $\mathbf{a}_{\text{ref}}$. In particular, this means that to account for scalability, measures will have to be taken to eventually allow a transition from a centralized database solution, as prescribed by P21, to one with caching facilities, as prescribed by P22.

At this point, for both end points of the scaling bandwidth (i.e., $\mathbf{a}_{\text{ref}}$ and $\mathbf{a}_{\text{max}}$) a physical distribution design has been made that ensures compliance with the original systems requirements, and within acceptable costs. In practice, additional points on the scale axis may have to be evaluated as well to ensure a transition from $\mathbf{a}_{\text{ref}}$ to $\mathbf{a}_{\text{max}}$.

By using AD-DIS the scalability of the application is evaluated while taking into account the scalability support of the execution environment. The scalability of the application is dealt with mostly at the logical level where distribution policies are actually chosen. These distribution policies are reflected in the design of the application.

The scalability support of the execution environment is primarily evaluated on the physical level. An execution environment providing lots of support for scalability offers several mechanisms to choose from. The differences between mechanisms should preferably be transparent to the application, as this will make scaling easier.

4 An Implementation Framework

AD-DIS supports a developer in distributing the functionality of an application, taking one or several scenario's into account. The result of applying AD-DIS are one or more distribution policies, which describe precisely which functionality should be distributed, and how and where that distribution should take place. However, the implementation of a distribution policy is still left open. For example, it may have been decided to allow replication of a database as long as the replicas adhere to strong consistency rules. In other words, clients should never be aware that the database

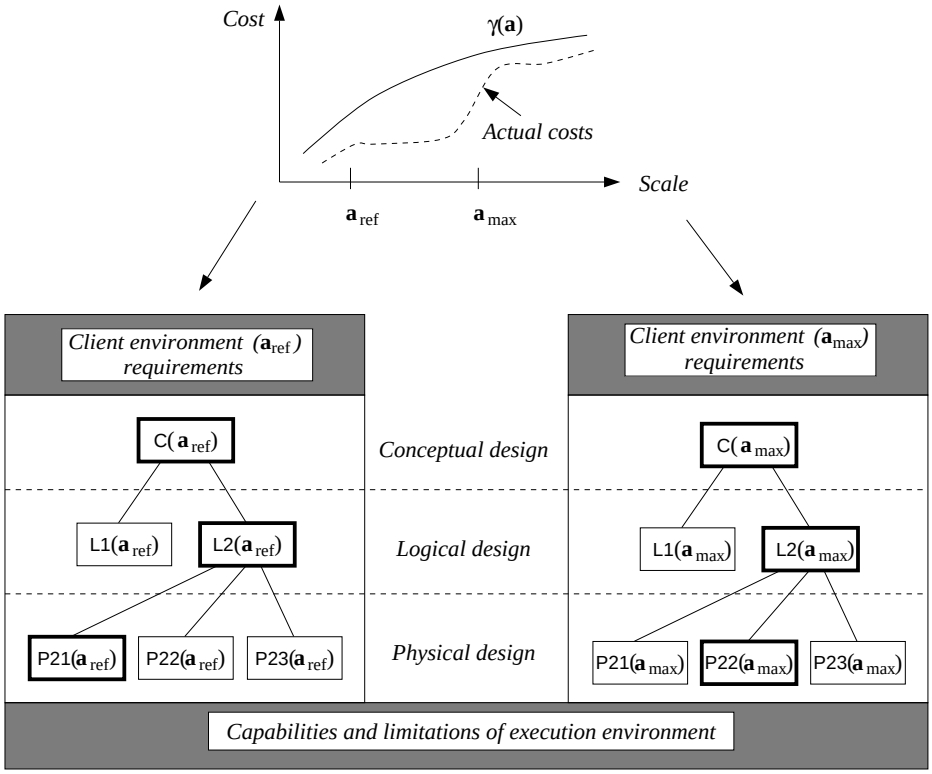

Figure 2: Designing for scalability using AD-DIS.

is physically distributed and replicated across several locations. However, the coherence protocol that implements such a distribution policy has not yet been defined, and indeed, several alternative implementations may exist.

4.1 Extensibility and Performance

Designing and implementing distribution policies in the face of scalability introduces several new aspects that have not yet been considered in the development process. Consider the following, more apparent ones.

Interoperability and heterogeneity. Modern applications should expect to be part of a large, enterprise-wide information system in which they have to interoperate with many other components. Scaling these information systems can be done only if their components are designed in such a way that they can indeed easily interoperate, but also be easily replaced and extended. One particular consequence is that we need to clearly separate interfaces from their implementation. We advocate an interface-based design approach by which a component is fully described by the services and interfaces it provides, as well as those it needs. The latter is also necessary to support different execution environments. By having the application specify or implement the same interfaces to the underlying platforms, portability, and thus extensibility is better supported.

Multiple administrative domains. Building large-scale, possibly wide-area applications implies that we may need to cope with multiple administrative domains. This places additional requirements on the management and security of an application.

First, it is necessary to integrate management support into the application. Otherwise, administrative domains will need management services that comprise knowledge concerning all applications they have to manage. With vast numbers of different applications, which may span multiple domains, such an approach is hard to maintain.

Second, security poses additional problems. On the one hand, parts of an application that reside in a specific administrative domain should comply to the security policies of that domain. On the other hand, an application should also be able to protect itself instead of having to rely on a domain's security policy. Like management, the consequence is that security should be partly integrated into an application.

Performance Issues. We would not be so much concerned about scalability if it did not affect the efficiency of systems. Unfortunately, keeping up performance is more easily said than done. In particular, we must anticipate that scaling techniques such as caching, replication, distribution, and parallelism need to be applied for mere performance reasons. However, it is important not to mix the implementation of scaling techniques with the implementation

of functionality. In fact, we claim that virtually all distribution issues can and should be separated from functionality.

Separation is important as we may need to switch to a different scaling technique when higher performance demands are required. Obviously, such demands should not affect the functionality of an application, and in particular its present implementation.

4.2 Distributed Shared Objects

A possible solution to enable scalable distributed applications is a wide-area distributed system called Globe [8]. Fundamental to the Globe system are **distributed shared objects**. In terms of AD-DIS, a distributed shared object models a nondistributable component as constructed in the logical distribution design. In other words, we assume the object's state cannot be further partitioned and subsequently distributed across multiple locations. However, it may be possible to *replicate* or *cache* that state.

Our objects differ from other distributed objects, including those used in popular systems like DCOM and CORBA, in two important ways. First, the state of a distributed shared object can be copied across multiple machines. Second, in contrast to existing systems, a distributed shared object fully encapsulates its own distribution policy. This means that all implementation aspects concerning, for example, the replication and migration of state, are completely hidden from clients.

In order for a client to invoke an object's method, it must first **bind** to that object. Binding results in an interface belonging to the object being placed in the client's address space, along with an implementation of that interface. Such an implementation is called a **local object**. This model is illustrated in Figure 3.

A local object resides in a single address space and communicates with local objects in other address spaces. Each local object is composed of several subobjects, and is itself again fully self-contained as also shown in Figure 3. A minimal composition consists of the following four subobjects.

Semantics subobject. This is a local object that implements (part of) the actual semantics of the distributed object. As such, it encapsulates the functionality of the distributed object. The semantics object consists of user-defined primitive objects written in programming languages such as Java or C++. These primitive objects can be developed independent of any distribution or scalability issues.

Communication subobject. This is generally a system-provided subobject. It is responsible for handling communication between parts of the distributed object that reside in different address spaces. Depending on what is needed from the other components, a communication subobject may offer primitives for point–to–point communication, multicast facilities, or both.

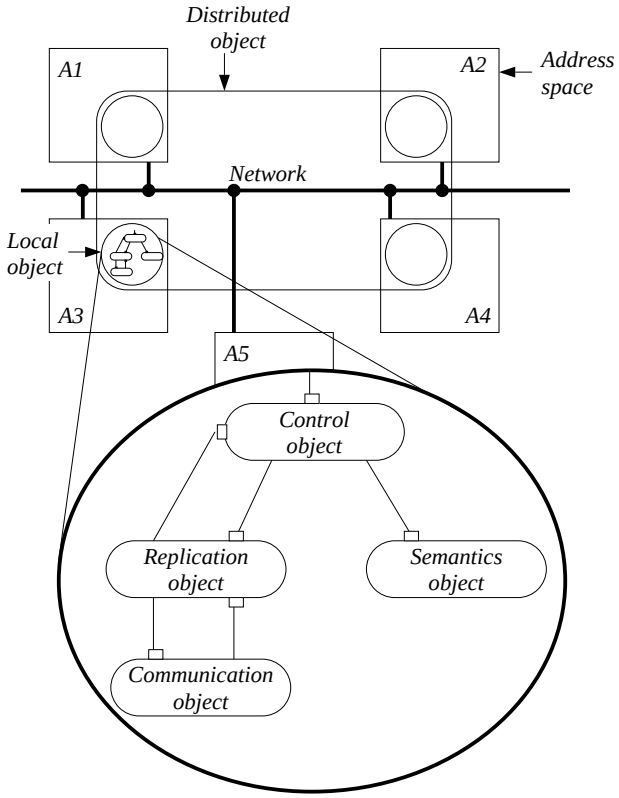

Figure 3: Example of an object distributed across four address spaces.

Replication subobject. The global state of the distributed object is made up of the state of its various semantics subobjects. Semantics subobjects may be replicated for reasons of fault tolerance or performance. In particular, the replication subobject is responsible for keeping these replicas consistent according to some (per-object) coherence strategy. Different distributed objects may have different replication subobjects, using different replication algorithms. An important observation is that the replication subobject has a standard interface. However, implementations of that interface will generally differ between replication subobjects. In a sense, this subobject behaves as a meta-level object comparable to techniques applied in reflective object-oriented programming.

Control subobject. The control subobject takes care of invocations from client processes, and controls the interaction between the semantics subobject and the replication subobject. This subobject is needed to bridge the gap between the user-defined interfaces of the semantics subobject, and the standard interfaces of the replication subobject.

A key role, of course, is reserved for the replication subobject. An important observation is that communication and replication subobjects are unaware of the methods and state of the semantics subobject. Instead, both the com-

munication subobject and the replication subobject operate only on invocation messages in which method identifiers and parameters have been encoded. This independence allows us to define standard interfaces for all replication subobjects and communication subobjects.

4.3 Globe and Scalability Problems

Obviously, distributed shared objects themselves do not provide scalable solutions, but instead, offer a scalable implementation framework in which such solutions can be incorporated. Current object-based frameworks or architectures do not provide this support. First, current distributed object models follow a traditional client/server approach in which the state of an object is confined to a single location at a time. Clients are offered only proxy implementations of an object's interface, which effectively implement a remote method invocation (RMI) mechanism. This is the only mechanism provided to access an object. Scalability techniques such as (transparent) caching and replication of state require more than just RMI.

Second, if distribution services are provided at all, they are specified external to objects. Consequently, a distribution service can apply only the same policy to all objects it manages. From a scalability point of view, such a one-size-fits-all approach is deemed to fail, for the simple reason that the effectiveness of a scaling or distribution policy is highly dependent on object-level semantics. Also, specifying services as separate entities still forces us to construct scalable solutions for those services. In other words, hardly any support is offered for developing scalable solutions.

Globe is different in this respect. The most important observation is that we recognize that solutions to scaling objects depend on the functionality of that object. For this reason, we advocate that solutions should be entirely encapsulated by an object. On the other hand, it is possible to separate the implementation of distribution policies from object-level semantics.

5 Conclusions

In this paper, we have argued that scalability requirements can be formulated precisely by providing a formal framework for expressing such requirements. However, more is needed to actually build scalable applications. First, we advocate that engineering methods such as AD-DIS should be used to assist a developer in deciding where and how data and functionality should be distributed. The engineering method must also provide a separation of concern regarding dealing with distribution *policies* in the application and distribution *mechanisms* in the execution environment. Second, we need an implementation framework in which functionality is separated from distribution issues, but which allows scalable solutions to be easily incorporated. Whether our formal framework, AD-DIS, and Globe are also generally applicable solutions towards building scalable applications, can only be demonstrated in practice. AD-DIS and Globe have been applied to practical situations, but more work is still needed.

In particular, we need more practical experience to actually see whether AD-DIS and Globe adequately support developing scalable solutions. Also, as it stands now, the three parts of our framework (requirements specification, design, and implementation) have not been integrated into a single methodology. Finally, our framework would improve if we could add design and implementation guidelines that could be derived from scalability requirements. Such guidelines are presently not available.

A major problem with building scalable solutions is that they are highly dependent on the application. Constructing general-purpose scalable solutions for replicating and distributing data and functionality, does not make much sense to our opinion. Consequently, reasoning about scalability in general becomes difficult. It is perhaps for this reason that so few papers exist on scalability. However, the approach discussed in this paper demonstrates that we can at least follow a more systematic and structured approach towards software engineering for scalable distributed applications.

References

[1] B. Neuman. "Scale in Distributed Systems." In T. Casavant and M. Singhal, (eds.), *Readings in Distributed Computing Systems*, pp. 463–489. IEEE Computer Society Press, Los Alamitos, CA., 1994.

[2] M. Schwartz and P. Tsirigotis. "Techniques for Supporting Wide Area Distributed Applications." Rep. CU-CS-519-91, Dept. Comp. Sc., Univ. Colorado, Feb. 1991.

[3] D. G. Messerschmidt. "The Convergence of Telecommunications and Computing: What are the Implications Today?." *Proc. IEEE*, 84(8):1167–1186, Aug. 1996.

[4] D. J. Kuck. *High Performance Computing*. Oxford University Press, New York, NY, 1996.

[5] A. Aue and M. Breu. "Distributed Information Systems: An Advanced Methodology." *IEEE Trans. Softw. Eng.*, 20(8):594–605, Aug. 1994.

[6] R. Wieringa. "A Survey of Structured and Object-Oriented Software Specification Methods and Techniques." *ACM Comput. Surv.*, 1998. *To appear*.

[7] S. van der Zijden, J. van 't Wout, and J. Schekkerman. "Architecture Design for Distributed Information Systems." *Informatie*, 38(10):62–68, Oct. 1996.

[8] M. van Steen, P. Homburg, and A. Tanenbaum. "The Architectural Design of Globe: A Wide-Area Distributed System." Rep. IR-422, Vrije Universiteit, Dept. Math. & Comp. Sc., Mar. 1997.