A Distributed Architecture Definition Language: a DADL

Ron Burback

December 18, 1998

Contents

List of Figures

List of Tables

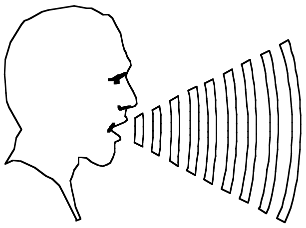
1 Abstract

Many computer science languages have been developed over the years that have concentrated on language fundamentals for the definition of algorithms and data structures. These traditional computer science languages give little help in defining the architecture of a system, especially a large distributed system. Architecture defines the components of a system and their interfaces, methods of communication, and behaviors. A Distributed Architecture Definition Language (DADL) is proposed that extends the existing paradigm used in programming to include architecture descriptions for a particular class of distributed systems architectures.

The architectural description language will provide fundamentals that concentrate on the conversation, communication, contracts, and behaviors of elements in the distributed system.

A DADL will be defined and used to describe a family of different distributive architectures. A program written in DADL can be compiled into different materializations of the architecture. Each materialization has different performance and resource characteristics leading to an optimizing choice.

It will be shown that large architectural variations can be described with minimal changes thus showing the elaboration tolerance of DADL programs.



2 Introduction

Distributed Architecture Definition Languages (DADLs) are emerging as tools for formally representing the architecture of distributed systems. As architectures become a dominant theme in large distributed system development, methods for unambiguously specifying a distributed architecture will become indispensable.

An architecture represents the components of a large distributed software system and their interfaces, methods of communication, and behaviors. It is the behaviors of the components, the communication between the pieces and parts, that are under-specified in current approaches. To date, distributed system architectures have largely been represented by informal graphics in which the components, their properties, their interaction semantics and connections, and behaviors are hand-waved in only partially successful attempts to specify the architecture.

Traditional computer languages, like C, concentrate mainly on the definition of the algorithm and data structure components by using language provided mechanisms to specify type definitions, functions, and algorithm control. The interface is under-defined by header files where function names, parameters, parameter types, and parameter order are specified. This is short of specifying the behavior of the interface. Traditional computer languages are much more suited to defining implementation than they are to defining architecture.

Consider the following simple C program where we calculate the sum of two integers. See Table 1 on page 8.

Traditional programming languages easily define the data structures and the algorithms. There is very limited help in defining the architecture. In fact, there is an assumed architecture, so implicit that most languages doesn't even define it as a feature. The functions *main* and *plus* communicate over a shared address space, memory resident, ordered, highly reliable, synchronous, and error-free communication medium materialized by using a call-frame stack.

The language of communication is defined by the *call* statement. The function *main* sends two integers to the function *plus* and waits for an integer in reply. The function *plus* receives two integers and replies with their sum. The implicit *call* and *return* in the C language materialize this architecture.

This implicit architecture is appropriate for small and simple programs but as applications become more complex, large, and distributed, the im-

The implementation file

```
#include <plus.h>
void main() {
    int results ;
    results = plus(1,2) ;
} ;

int plus (int n , int m) {
    return n+m ;
} ;
```

The header file

```
int plus (int n , int m) ;
```

Table 1: Example of a Traditional C Program with Header File.

plicit call-frame stack architecture is no longer appropriate. A distributed architecture might deal with a disjoint address space, non-memory resident, unordered, non-reliable, asynchronous, and error prone communication mechanism. This is far from the assumptions of traditional computer languages. It is no wonder that large systems are hard to define using traditional programming languages.

Object based systems, like C++, extend the programming paradigm to include objects, sub types, polymorphism, and inheritance. This powerfully extends the ability of a language to define the data structures and algorithms. However, the underlying implicit architecture does not change. The architecture still dictates memory resident, ordered, highly reliable, synchronous, and error free communication over a shared address space, that is materialized by a call-frame stack.

Another shortfall of the implicit object-based system architecture is in the definition of the behavior. Though the C++ interface defines the methods exported by a class, it does not define the methods used or required by that class. Thus an implementation can perfectly match the interface but have an entirely different behavior than another similar implementation because it is composed with different primitives.

Some of the founding object-based languages, such as SIMULA [KO62] and SmallTalk [GAR83] [KG83] [GA84] [GAR89], tried to replace the implicit architecture of a call-frame stack with a message-passing queue. In this architecture, methods are evoked by passing messages between objects. However, the architecture is still implicit and under-defined leading to no choice in an alternative behaviors.

Distributed middle ware support systems, like DCE [DCE96], extend the programming paradigm. The DCE Interface Definition Language(IDL) includes argument flow (in or out parameters), interface identifiers, dynamic binding information, and exceptions. Using DCE it is possible to define communication mechanisms for architectures that are in disjoint address spaces, non-memory resident, non-reliable, and error prone. DCE accomplishes this by expanding the *call* mechanism. Asynchronous communication is dealt with by providing threads while unordered communication is provided by using network data grams under UDP. DCE replaces the traditional architecture with one that is more suited for distributed computing but does not allow choice between alternative architectures.

CORBA [COR96] extends the programming paradigm to include messaging and distributed objects. Communication is done over an information bus where requests are issued and brokers respond to satisfy those requests. CORBA is really directed at building object models for a large class of applications under one, and only one, request/broker architecture. Though this is extremely necessary for application development, architectural needs go unfulfilled. CORBA is more like a detailed requirement specification, defining in detail the needs of a particular application domain.

Megaprogramming [WWC92] extends the call mechanism to an asynchronous messaging paradigm between large components called megamodules. The communication between two megamodules is defined with language structures like setup, estimate, invoke, extract, and examine.

Languages, like Rapide [LV96], extend the interface definitions to include events and causal relationships between events. Using the paradigm of hardware design, the behavior of the interface is governed by signals and events which are synchronized by a clock. The interface has been extended to include both the generated and required methods. This allows for the interface to act more like a meta-schema that governs both actions and simple behavior.

In comparison, Rapide expands the role of the call statement into a directed graph of causal events. Megaprogramming expands the call statement into a family of asynchronous primitives. While the proposed DADL expands the call statement into conversations, behaviors, and contracts concentrating on distributed systems.

Don't confuse a DADL with a requirement language. The requirement is a statement of the problem at a high level of abstraction. This is in contrast to a DADL which defines a generic plan that binds the requirements to the implementation. Requirement languages, such as STATEMATE [HLN+88] and Modechart [JM94], define the problem but not the solution.

This thesis proposes a DADL to specify architectures of distributed systems. This is accomplished by first defining the attributes of large distributed systems that distinguish a distributed system of other types of systems. Next the DADL language will be defined. DADL will then be used to specify several key architectures.

Other related work includes Rapide [LV96], UniCon [SDK+94], ArTek [HRCP94], Wright [AG94], Code [NB92], Demeter [PXL95], Modechart [JM94], PSDL/CAPS [LSBH93], Resolve [EHL+94] and Meta-H [Ves94a].

3 Distributed Architecture Description Language

3.1 Overview

There are two main, tightly-coupled concepts in DADL which are the conversation and the participants. The conversation has an identity independent of the participants. Contracts govern the behavior of the conversation and the expectations of the participants. A conversation is of little use without participants. Likewise, participants are of little use without communication taking place in a conversation.

The participants produce and consume the conversation. These participants are distributed agents, or simply dagents.

Dagents communicate with other dagents to accomplish tasks. A dagent may be involved in many conversations, potentially entering and leaving a conversation at times. The identity of the dagent at the other end of the conversation may be unknown.

For each conversation, a domain-specific language is defined. The base level unit of a communication is a term. Terms are built from characters taken from an alphabet. Multiple terms from the same dagent form a sentence while multiple sentences between dagents form a conversation. The conversation uses one alphabet.

The conversation is carried over a bi-directional connection. Information can be placed on a connection from many sources. The dagents may be unaware of the origin of the information. What happens to the information after it is placed on the connection is governed by the conversation manager.

The connection may be multi-sided in which case many dagents may get involved in the conversation.

A dagent may provide value by performing some service, or function, while the conversation is taking place and contributing the results back to the conversation.

The dagents agree to a contract to determine their behavior and expected behavior of all other agents participating in the conversation. A dagent may provide many services to the conversation.

Both the conversation and the dagents exist in an environment. From the environment, resources are provided and consumed.

The services that a dagent provides, the underlying definition of the conversation, along with the contract collectively define the architecture within an environment.

3.2 Backus-Naur Form (BNF) Semantics

The following sections we will define the language along with the supporting concepts.

The BNF semantics used are as follows:

1. Words inside double quotes ("word") represent literal words themselves (these are called terminals). The phrase double_quote is used to represent the double-quote character itself. An example literal: "if".
2. Words outside double quotes (possibly with underscores) represent syntactic categories (i.e. nonterminals). An example nonterminal: if_statement.
3. Syntactic categories are defined using the form: syntactic_category ::= definition
4. Square brackets ([ ]) surround optional items.
5. Curly brackets ({ }) surround items that can repeat zero or more times.
6. A vertical line (|) separates alternatives.

3.3 Conversations

See Table 2 on page 13.

conversation_block	::=
	"conversation"
	conversation_identifier
	"{"
	connection_block
	dagent_block
	alphabet_block
	term_block
	sentence_block
	converstion_behavior_block
	conversation_contract_block
	conversation_option_block
	"}"
	";"

Table 2: Basic Conversation Block

A conversation contains a list of connections. The content of the conversation are defined over an alphabet using terms and sentences. The well-formed sentences are described by the behavior while the performance is defined in the contract. The conversation has a collection of options.

3.3.1 Conversation Identifier

A conversation identifier is a unique string representing the name of the conversation. This name can then be used as a placeholder to indicate the conversation itself.

3.4 Connections

See Table 3 on page 14.

connection_block	::=	connection | connection_block
connection	::=	"connection" connection_identifier "(" connection_option_block connection_contract_block ")" ";"
connection_option_block	::=	"1-to-1," | "1-to-M," | "M-to-1," | "M-to-M,"
connection_contract_block	::=	connection_contract_list
connection_contract_list	::=	connection_contract_order connection_contract_delivery
connection_contract_order	::=	"ordered FIFO," | "ordered LIFO," | "unordered,"
connection_contract_delivery	::=	"guaranteed delivery " | "no guaranteed delivery"

Table 3: Connection Block

A connection is the interface of a conversation to the dagents. Information comes into a conversation over a connection and information is returned to dagents over a connection. The particular dagents, at any one time, talking or listening on a connection, may be dynamic. A connection is bi-directional and may have performance requirements governed by a contract.

3.4.1 Connection Identifier

A connection identifier is a unique string representing the name of the connection. This name could then be used as a placeholder to indicate the connection itself.

3.4.2 Connectivity

The connection may have only one producer and one listener. If so, then the connection is said to be 1-to-1. In many cases, this would imply one dagent.

The connection may have one producer and many listeners. If so, then the connection is said to be 1-to-M. A broadcasted message from one dagent to many dagents is one special case of a 1-to-M connection.

The connection may have many producers and only one listener. If so, then the connection is said to be M-to-1.

The connection may have many producers and many listeners. If so, then the connection is said to be M-to-N. Here information from many dagents can be directed to many other dagents.

3.4.3 Order

A connection may be ordered in which case the information placed on the connection will be available to the conversation in the same order. The transport layer in the traditional network model is an ordered connection with quarantined delivery.

The information on the connection could be used as FIFO (a queue) or LIFO (a stack).

3.4.4 Delivery

Once information is placed on the connection delivery may be guaranteed. Some connection may not guarantee delivery in which case information may be lost. The UDP layer in the traditional network model is an example of an unordered connection with no guaranteed delivery.

3.5 Dagent

See Table 4 on page 16.

dagent_block	::=
	"dagent"
	dagent_identifier
	"on"
	connection_identifier
	";"

Table 4: Dagent Block

3.5.1 Dagent Identifier

A dagent identifier is a unique string representing the name of the dagent. This name could then be used as a placeholder to indicate the dagent itself.

3.5.2 Dagent Connections

A dagent is associated with a connection. Recall that the connection is bi-directional. The dagent can both speak and listen on the connection.

3.6 Alphabet

See Table 5 on page 17.

alphabet_block	::=	
		"alphabet"
		"("
		"ASCII" | "EBSIDEC" | "BCD" | "BYTE"
		")"
		";"

Table 5: Alphabet Block

The lowest component of the conversation is the alphabet. For any given conversation the choice of alphabet is fixed. The choices might include ASCII, EBSIDEC, BCD, or uninterpreted 8-bit bytes. One element in the alphabet is called a character.

3.7 Terms

See Table 6 on page 19.

A group of characters is called a term. Terms may represent constants, variables, or functions. A term is the lowest element spoken by a dagent.

3.7.1 Term Identifier

A term identifier is a unique string representing the name of the term. This name could then be used as a placeholder to indicate the term itself.

3.7.2 Type Identifier

The type identifiers are borrowed from other languages and might include integers, floating point, character strings, booleans, constant expressions, and arbitrary large objects called blobs.

3.7.3 Constant Term

A grounded value of a particular type is a constant term.

3.7.4 Variable Identifier

A variable identifier is a unique string representing the name of the variable. This name could then be used as a placeholder to indicate the variable itself.

3.7.5 Variable Term

A term which has a known type but the value is determined dynamically during the conversation and may vary is called a variable.

3.7.6 Function Term

Function terms are explained in more detail in the next section.

term_block	::=	
	term_definition | term_block	
term_definition	::=	
	"term"	
	term_identifier	
	"("	
	constant_term | variable_term | function_term	
	")"	
	";"	
constant_term	::=	
	type_identifier	
	constant_value	
variable_term	::=	
	type_identifier	
	variable_identifier	
function_term	::=	
	type_ identifier	
	function_identifier	
	"("	
	parameter_list	
	")"	
	function_contract	
parameter_list	::=	
	variable_term | "," parameter_list	

Table 6: Term Block

3.8 Function Term

See Table 7 on page 21.

A function term represents a service provided by a dagent. This puts a restriction on the class of dagents able to emit this term. Only dagents which provide the function or service can participate in the conversation.

3.8.1 Function Identifier

A function identifier is a unique string representing the name of the function. This name could then be used as a placeholder to indicate the function itself.

3.8.2 Parameter List

A parameter list of the arguments to the function.

3.8.3 Function Quality of Service Contract

A function can have several different profiles with respect to the quality of service.

A contract may govern the acceptable error rate. One of the underlying assumptions that distinguishes distributed systems from more traditional systems is that individual elements in the system may fail without total system failure. Many of these failures are masked from the system, itself, by using replication and distribution.

An element in a system can provide many different levels of service. This is part of the behavior of that element. The levels include:

Errors Tolerated In the case of tolerated errors, the service representing the function may be unavailable at numerous times. A single service on a single computer would be an example of an errors tolerated quality of service. If the machine goes down, the service is unavailable.

The service is available most of the time but on occasions errors are known to happen. The exact threshold of error tolerance, the number of errors per unit of time, is service specified. There may have to be manual intervention to correct from an error state.

<table>
<tr><td>function_term</td><td>::=
type_ identifier
function_identifier
“(”
parameter_list
“)”
function_contract</td></tr>
<tr><td>parameter_list</td><td>::=
variable_term | “,” parameter_list</td></tr>
<tr><td>function_contract</td><td>::=
function_quality_of_service
function_performance</td></tr>
<tr><td>function_quality_of_service</td><td>::=
“errors tolerated,” | “highly available,” | “fault tolerant,”</td></tr>
<tr><td>function_performance</td><td>::=
“performance” | “latency” | “error rate”
“(”
“min” “=” <integer>
“max” “=” <integer>
“avg” “=” <integer>
“rms” “=” <float>
“)”
“certainty” <integer>
“ ” | “,” function_performance</td></tr>
</table>

Table 7: Function Term Block

Highly Available The service representing the function will be available a significant amount of the time but occasionally, the service may experience short periods of unavailable.

An example of such a quality of service would be a single service replicated on many machines. Many machines may go down but the service remains available. Of course, if all machines go down, service will not be available.

This level has one distinguishing characteristic of a known error free state. In a potentially automated fashion, the service in error is reset to the known error free state, usually an initial state, and restarted. The total distributed system continues to operate with what looks to the users as a minor event.

Highly available services are usually one or two point fault tolerant.

Fault Tolerant A fault tolerant service requires at least triple redundancy. At least three machines are running identical services on identical parameters. The results are then compared. If all three results are identical, the result is returned. One machine can go down in this configuration without effecting the results. Hot spares are available to replace a faulty machine and join an operational machine pair.

If a service is guaranteed to be error free it is said to be fault tolerant. Ideally, one would wish all services to be fault tolerant but this is an expensive capability and demands large quantities of resources to obtain this level. Only critical services need to be fault tolerant. Most systems work well at highly available levels.

3.8.4 Function Performance Contract

The performance of a function is stochastic in nature. For example, the average performance is only achieved a certain percentage of time.

Performance This represents the number of function invocations per second.

Latency This represents the time from initialization of a function until the first information is available.

Error Rate This represents the number of times a function fails per second. This is not a semantic failure where the incorrect results are returned but the number of times no result is returned at all.

Certainty The percentage of time that one of the performance parameters is achieved.

3.9 Sentences

See Table 8 on page 24.

sentence_block	::=	
	sentence_definition | sentence_block	
sentence_definition	::=	
	"sentence"	
	sentence_identifier	
	"("	
	term_list	
	")"	
	"from"	
	conenction_identifier	
	"to"	
	connection_identifier	
	";"	
term_list	::=	
	term_identifier | "," term_list	

Table 8: Sentence Block

A list of terms form a sentence. Every sentence is generated by only one dagent but, depending on the connections in the conversation, may be consumed by many dagents. The sentence enters the conversation from one connection and leaves the conversation at another connection.

In the conversation the sentence is atomic. The sentence is either all present or not present at all. A sentence cannot be interrupted preventing the overlapping of two sentences.

Every sentence has an identifier.

3.10 Conversation Behavior

See Table 9 on page 26.

A conversation has many sentences. The sentence order determines the behavior. This order could be very simple or quite complex.

3.10.1 Sequential

Two sentences are sequential if they enter the conversation one after the other. This is indicated with a ";".

3.10.2 Parallel

Two sentences are parallel if they can enter the conversation at the same time. This is indicated with a "|".

3.10.3 Wait

If more than one sentence is coming into the conversation in parallel and the conversation needs to block or wait to continue until all sentences are available, then this is accomplished with a sentence join. This is indicated with a "wait".

3.10.4 At Least Once

With this option, a sentence can be repeated one or more times. This is indicated with a "+".

3.10.5 Zero or More Times

With this option, a sentence can be repeated zero or more times. This is indicated with a "*".

converstaion_behavior_block	::=
	“behavior”
	“(”
	compound_fragment
	“)”
	“;”
compound_fragment	::=
	sentence_identifier
	“(” sequential_fragment “)”
	|
	“(” parallel_fragment “)” “wait”
	|
	“(” repeat_fragment “)”
sequential_fragment	::=
	compound_fragment
	|
	compound_fragment “;” sequential_fragment
parallel_fragment	::=
	compound_fragment
	|
	compound_fragment “|” parallel_fragment
repeat_fragment	::=
	compound_fragment [“+” | “*”]

Table 9: Conversation Behavior Block

3.11 Contract

See Table 10 on page 27.

conversation_contract_block	:: =
	“contract”
	“performance” | “latency” | “error rate” | “capacity”
	“(”
	“min” “=” <integer>
	“max” “=” <integer>
	“avg” “=” <integer>
	“rms” “=” <float>
	“)”
	“;”
	[conversation_contract_block]

Table 10: Contract Block

3.11.1 Performance

This represents the number of conversations per second.

3.11.2 Latency

This represents the time from initialization of a conversation until the first information is available.

3.11.3 Error Rate

This represents the number of times a conversation fails per second.

3.11.4 Capacity

This represents the number of simultaneous conversations that can occur within one conversation manager.

3.11.5 General Observations about a Contract

A contract is a general concept for characterizing and regulating cooperation in the architecture. A contract is an agreement that governs the cooperation and embodies the obligation and expectation associated with cooperation. Needless to say, agreement by members of the distributed system to a contract, enables cooperation to take place.

The contract based approach to software development includes two pieces. The contract first specifies what the consumer must do and secondly, the contract specifies what the supplier will provide. The contract does not specify how the supplier and consumer will fulfill the contract. There is not much support from commercial programming languages for contract concepts with Eiffel [Mey92] being one of the few exceptions.

The contract is an abstraction of the behavior of the system, the interactions under constraints, and the interface. The contract defines the obligations, permissions, and prohibitions of the elements in the system.

A contract can be placed on many different kinds of elements. These include dagents, connections, conversations, interface, services, servers, resources, the environment, functions, and procedures.

A contract might include constraints on the bandwidth and latency. Traditionally, these are measurements associated with data through a channel, but these metrics can be used to measure how many times per second a service can be evoked (the generalized bandwidth) and the time it takes before the first service returns an answer (the generalized latency). These constraints might include minimum, maximum, and average expectations.

3.12 Conversation Options

See Table 11 on page 29.

conversation_option_block	::= "with" "options" "(" conversation_option_list ")" ";"
conversation_option_list	::= "persistence" | "volatile" "," "data open" | "data integrity" | "data privacy" "," "marshaled" | "unmarshaled" "," "authenticated" | "unauthenticated" "," "authorized" | "unauthorized"

Table 11: Conversation Options Block

There are many different kinds of conversations. A conversation may be one-sided or it may be a peer-to-peer interchange. A conversation may include a number of parties sharing information.

3.12.1 Persistence

If the conversation can recover from a total system environment crash, then it is persistent. Persistent information require more resources to establish and accomplish but give better total-system error recovery capability. A conversation with control structures in a transaction log on disk is an example.

3.12.2 Volatile

If the conversation is ephemeral and not recoverable after a system failure, then the conversation is said to be volatile. A conversation with control structures in RAM is an example.

3.12.3 Data Open

If the data can be viewed by a casual observer, then the data is said to be open.

3.12.4 Data Integrity

If the data can be verified to be correct with a digital signature mechanism, then the data is said to have integrity.

3.12.5 Data Privacy

If the data cannot be viewed by a casual observer, then the data is said to be private.

3.12.6 Marshaled

If the data is converted between alternative underlying representations without losing significant information, then the data is said to be marshaled.

3.12.7 Authenticated

If the identify of the dagents in a conversation can be verified, then the conversation is said to be authenticated.

3.12.8 Authorized

If an authenticated dagent has permission to interact with the conversation, then the dagent is said to be authorized.

3.13 Dagent

See Table 12 on page 31.

dagent_definition	::=
	"dagent"
	dagent_identifier
	"{"
	connection_block
	obeys_block
	dagent_contract_block
	resource_block
	service_block
	"}"
	";"

Table 12: Dagent Block

A dagent is similar in concept to the more familiar terminology of agent. However, to avoid confusion in overloading of definitions of the term agents from other fields, agents in DADL will be called dagents. One may think of a dagent as a process on a particular computer. There can be many processes on any one particular computer and each process may be multi-threaded.

The dagent may perform multiple functions and procedures called services. If the dagent is programmed with the traditional object paradigm then the inheritance mechanism can be used to provide polymorphism.

The behavior of a dagent is governed by a collection of actions combined with a set of constraints on when those actions may occur including internal actions. The actions that take place are restricted by the environment.

A dagent uses resources. Some of these resources are consumed and never to be used again while other resources are only temporarily allocated to the dagent and then recycled. The resources are provided by the assumed underlying environment.

A dagent may have state with events that change state. Dagents may have an error state caused by an internal fault. A dagent can be created, killed, and replicated. A dagent, though ready to execute, may be temporarily paused.

An example of a dagent could be a math process providing the simple arithmetic services including plus, minus, multiplication, and division.

3.13.1 Connection

The connection block for a dagent is the same as the connection block for a conversation.

3.13.2 Obeys

See Table 13 on page 33.

obeys_block	::=
	"obey conversation"
	conversation_identifier
	"as"
	dagent_identifier
	";"

Table 13: Obeys Conversation Block

The dagent inherits much information from the conversation. This is required by the obeys block.

3.13.3 Contract

The contract is the same as in the conversation.

The dagent's contract must be compatible with the conversation contract.

3.13.4 Resource

See Table 14 on page 35.

```
resource_block  :: =
                “resource”
                “shared” | “private”
                “disk” | “RAM” | “CPU” | “locks” | “binary” | “vm”
                “(”
                “min” “=” <integer>
                “max” “=” <integer>
                “avg” “=” <integer>
                “rms” “=” <integer>
                “)”
                “;”
                [ resource_block ]
```

Table 14: Resource Block

A dagent has associated resources that it may consume or use temporally and then recycle. Some resources may be shared with other dagents. Resources might include CPU, RAM, disk space, locks, and other peripheral devices like cdrom, tape drives, and printers. Dagents may share address space and name space with other dagents.

3.13.5 Service

See Table 15 on page 36.

service_block	::=
	function_term [service_block]

Table 15: Service Block

The service block list the number of services offered by the dagents. This list must be compatible with the conversations it joins. This is enforced at run time by the conversation manager.

The function list is the same as in the conversation.

3.14 The Environment

The environment is the part of the system that is assumed to be present. The environment contract defines the temporal constraints (deadlines), throughput requirements, availability, reliability, maintainability, security, safety, location, and quality of service.

4 Using DADL

See Figure 1 on page 38 for a diagram on DADL development.

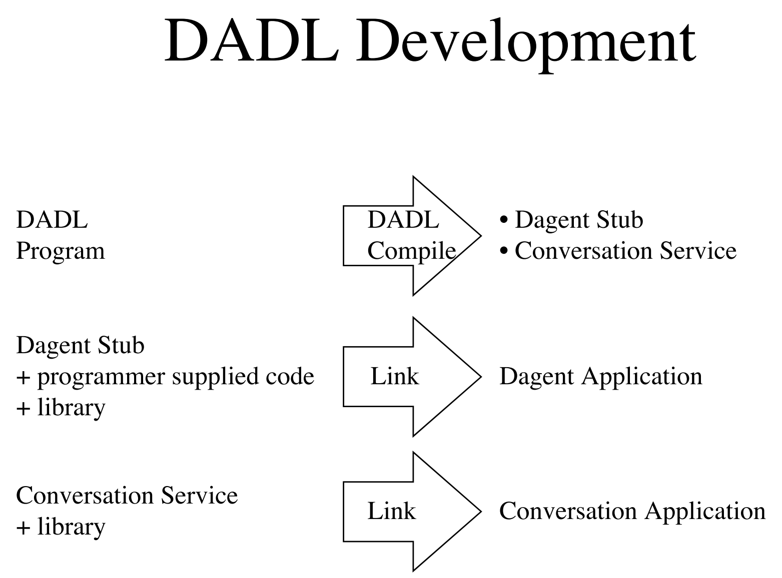


Figure 1: DADL Development

The customer programs a distributed architecture in the DADL language. The DADL compiler emits stub code for each dagent and for the conversation.

The dagent stub code contains the necessary components to communicate with the conversation. In addition, the dagent stub code includes function templates. These templates only include the call interface as defined by the services in the architecture. These templates are then completed by the customer programmer to provide the runtime content. The remaining dagent code is provided by the writer of the dagent. To aid the dagent programmer is a run time dagent library.

The conversation code contains the necessary components to materialize the conversation including the connection interfaces to the dagent. The conversation code manages the conversation, monitors the behavior, and enforces the contracts. A run time conversation library aids the conversation.

See Figure 2 on page 39 for diagram on DADL development

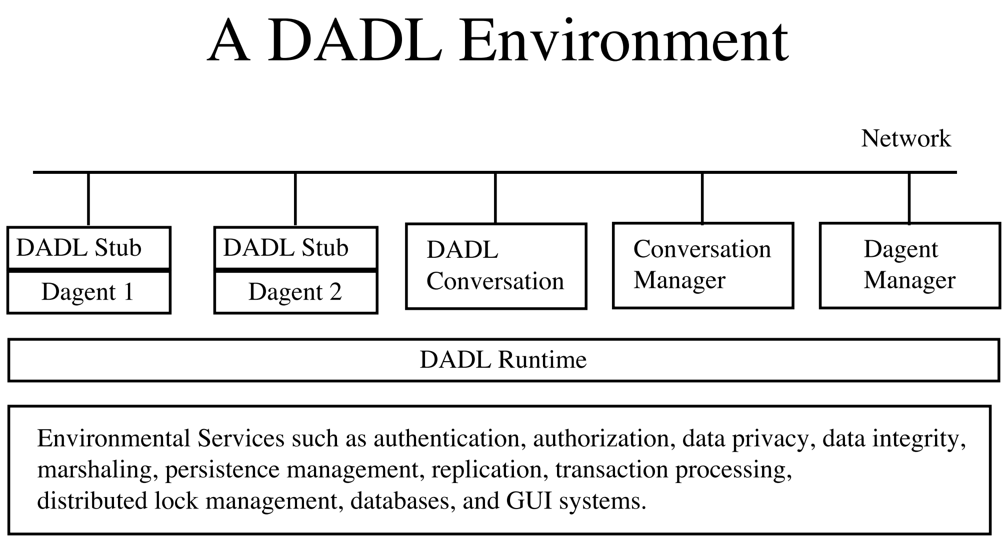


Figure 2: DADL Network Environment

The environment must contain two major services: namely the conversation manager and the dagent manager.

The conversation manager creates, destroys, and initializes conversations as well as managing the dagents entering and leaving the conversation. Special properties of the conversation such as persistence and reliability are enforced.

The dagent manager creates, destroys, initializes, migrates, pauses, and restarts dagents. Special properties of the dagents such as replication are enforced.

The DADL support environment is assumed to be robust. This includes services for authentication, authorization, data privacy, data integrity, marshaling, persistence management, replication, and service groups. Transaction processing, distributed lock management, databases, and GUI systems are assumed present.

5 Theory

5.1 The Theorem

A DADL can be defined and used to describe a family of different distributive architectures. A program written in DADL can be compiled into different base-level materializations of the architecture. Each materialization has different performance and resource characteristics leading to an optimizing choice.

5.2 The Proof Outline

1. Define DADL and the family of distributed architectures and then show that the distributed architectures can be expressed by a DADL program.

2. Define base-level target materialization including their performance and resource characteristics.

3. Compile and optimize the DADL program into the base-level target materializations.

6 Examples

6.1 Simple Client Server

This is an example of a simple client server application where the call statement is replaced with a simple communication language.

The service provided by the server is "plus", in which two parameters are added together generating the resultant.

The architecture reflects the architecture assumed in many traditional programming langauges.

6.1.1 Interface

```
conversation simple_plus {
      connection c1 (1-to-1, ordered FIFO, guarenteed delivery) ;
      connection c2 (1-to-1, ordered FIFO, guarenteed delivery) ;

      dagent d1 on c1 ;
      dagent d2 on c1 ;

      alphabet ( BYTE ) ;

      term t1 ( int n ) ;
      term t2 ( int m ) ;
      term t3 ( int plus ( int n , int m ) highly available ) ;

      sentence s1 (t1, t2) from c1 to c2 ;
      sentence s2 (t3) from c2 to c1 ;

      behavior (s1 ; s2) ;
      with options (volatile, data open, unmarshaled, unauthenticated, unauthorized);
} ;
```

6.1.2 Dagent Client

```
dagent main {
    connection c1 (1-to-1, ordered FIFO, guaranteed delivery) ;
    obeys conversation simple_plus as d1 ;
    resource shared ram (max = 64,000) ;
} ;
```

6.1.3 Dagent Server

```
dagent plus {
    connection c2 (1-to-1, ordered FIFO, guarenteed delivery) ;
    obeys conversation simple_plus as d2 ;
    resource shared ram (max = 64,000) ;
    int plus ( int n , int m ) highly available ;
```

6.2 Three Tier Client Server

This is an example of three tier client server application. The presentation client calls a middle layer which passes the information onto a third, server layer for computation. There are three dagents and two conversations. The middle dagent participates in two conversations.

6.2.1 Interface, Presentation to Application

```
conversation presentation_application {
    connection c1 (1-to-1, ordered FIFO, quaranteed delivery) ;
    connection c2 (1-to-1, ordered FIFO, quaranteed delivery) ;

    dagent d1 on c1 ;
    dagent d2 on c2 ;

    alphabet ( BYTE ) ;

    term t1 (int n) ;
    term t2 (int m) ;
    term t3 (int results) ;

    sentence s1 (t1, t2) from c1 to c2 ;
    sentence s2 (t3) from c2 to c1 ;

    behavior (s1 ; s2) ;

    with options (volatile, data open, marshaled, unauthenicated, unauthorized);
} ;
```

6.2.2 Interface, Application to Server

```
conversation application_server {
      connection c1 (1-to-1, ordered FIFO, quaranteed delivery) ;
      connection c2 (1-to-M, ordered FIFO, quaranteed delivery) ;

      dagent d1 on c1 ;
      dagent d2 on c2 ;

      alphabet ( BYTE ) ;

      term t1 (int n) ;
      term t2 (int m) ;
      term t3 ( int plus ( int n , int m ) highly available ) ;

      sentence s1 (t1, t2) from c1 to c2 ;
      sentence s2 (t3) from c2 to c1 ;

      behavior (s1 ; s2) ;

      contract performance (min=1, max=2, avg=1.1, rms=.1) ;
      contract latency (min=10,max=20,avg=15,rms=3) ;
      contract error (min=0,max=0,avg=0,rms=0) ;
      with options (volatile, data open, marshaled, unauthenicated, unauthorized);
} ;
```

6.2.3 Dagent Presentation

```
dagent main {
       connection c1 ;
       obeys conversation presentation_application as d1 ;
       contract performance (min=1,max=2,avg=1.1,rms=.1) ;
       contract latency (min=10,max=20,avg=15,rms=3) ;
       contract error (min=0,max=0,avg=0,rms=0) ;
       resource (ram=64000, disk=0, vm=120000,
binary=main.exe) ;
};
```

6.2.4 Dagent Application

```
dagent application {
       connection c1 ;
       connection c2 ;
       obeys conversation presentation_application as d2;
       obeys conversation application_server as d1;
       contract performance (min=1,max=2,avg=1.1,rms=.1) ;
       contract latency (min=10,max=20,avg=15,rms=3) ;
       contract error (min=0,max=0,avg=0,rms=0) ;
       resource (ram=64000, disk=0, vm=120000,
binary=application.exe) ;
};
```

6.2.5 Dagent Server

```
dagent plus {
    connection c1 ;
    obeys conversation application_server as d2 ;
    contract performance (min=1,max=2,avg=1.1,rms=.1) ;
    contract latency (min=10,max=20,avg=15,rms=3) ;
    contract error (min=0,max=0,avg=0,rms=0) ;
    resource (ram=64000, disk=0, vm=120000, binary=plus.exe) ;
    int plus ( int n , int m ) highly available ;
```

6.3 Simple email

This is the heart of simple email. Most of the work is in the communication layers the dagents are quite simple.

6.3.1 Email Communication

```
conversation email {
    connection c1 (1-to-1, ordered FIFO, quaranteed delivery) ;
    connection c2 (1-to-1, ordered FIFO, quaranteed delivery) ;

    dagent d1 on c1 ;
    dagent d2 on c2 ;

    alphabet ( ASCII ) ;

    term t1 (const "MAIL FROM: ") ;
    term t2 (string from_user ) ;
    term t3 (const "250 OK") ;
    term t4 (const "RCPT TO: " ) ;
    term t5 (string to_user ) ;
    term t6 (const "DATA" ) ;
    term t7 (blob the_message ) ;
    term t8 (const "<CRLF>.<CRLF>" ) ;

    sentence s1 (t1, t2) from c1 to c2 ;
    sentence s2 (t3) from c2 to c1 ;
    sentence s3 (t4, t5) from c1 to c2 ;
    sentence s4 (t6) from c1 to c2 ;
    sentence s5 (t7 ) from c1 to c2 ;
    sentence s6 (t8 ) from c1 to c2 ;

    behavior (s1 ; s2 ; s3 ; s2 ; s4 ; s5+ ; s6 ) ;

    contract performance (min=1,max=2,avg=1.1,rms=.1) ;
    contract latency (min=10,max=20,avg=15,rms=3) ;
    contract error (min=0,max=0,avg=0,rms=0) ;
```

with options (volatile, data open, marshaled, unauthenicated, unauthorized);

}

6.3.2 Email Client

```
dagent client {
      connection c1 (1-to-1, ordered FIFO, quaranteed delivery) ;
      obeys conversation email as d1 ;
      contract performance (min=1,max=2,avg=1.1,rms=.1) ;
      contract latency (min=10,max=20,avg=15,rms=3) ;
      contract error (min=0,max=0,avg=0,rms=0) ;
      resource (ram=64000, disk=0, vm=120000,
binary=elm.exe) ;
};
```

6.3.3 Email Server

```
dagent server {
      connection c1 (1-to-1, ordered FIFO, quaranteed delivery) ;
      obeys conversation email as d2 ;
      contract performance (min=1,max=2,avg=1.1,rms=.1) ;
      contract latency (min=10,max=20,avg=15,rms=3) ;
      contract error (min=0,max=0,avg=0,rms=0) ;
      resource (ram=64000, disk=0, vm=120000,
binary=pop.exe) ;
};
```

7 Distributed System Foundations

Distributed systems exhibit fundamental inherent characteristics that distinguishes them from more traditional single-computer, single-user, single-process systems.

The elements of a distributed system become less dependent on a particular programming language, the hardware instruction set, and the underlying operating system but more dependent on the style of communications, interfaces, and the behavior exhibited during those interactions.

7.1 Distribution

The most obvious characteristic of a distributed system is distribution. A distributed system consists of many computers, distributed over many location, with remote communications. There are many different kinds of remote communications in a distributed system ranging from remote access of terminal devices to dynamically allocated IPs of mobile computer systems.

There may be distributed data access from one computer to another. Data may be vertically or horizontally distributed across many computers and disk farms. Peripheral devices may be network mounted allowing access from many locations. The application may be divided into many functional layers, or tiers, with each tier potentially on different computer systems thus building the popular client-server application architecture. Parallel algorithms may be distributed across many systems.

Distribution is a general terminology which includes the case of a occasionally connected computer or nomadic computing where systems frequently are disconnecting and rejoining the environment. New partitions are being formed on a regular bases affecting data, locks, networks, and applications. Dynamic changes in the routing tables are needed as mobile IPs become available to the general user. A disconnected customer must still be able to work, though in a limited fashion, using a self contained environment provided on his personal machine. When the disconnected system rejoins the distributed system environment it will have to be brought up to consistent status with the rest of the environment. This might include data, messages, locks, and application updates. Asynchronous and persistent messaging as well as data caches become more important in supporting the partitioning demanded by nomadic computing.

7.2 Concurrent

Many, if not all, components in a distributed system work independently, in parallel, with little interaction. For the most part, many of the systems in a distributed system work autonomously only communicating with a small collection of other components at interaction points.

An application written for a distributed computing environment may have many concurrent threads of operations.

Computers in a highly parallel machine are similar but not the same as computers in a distributed system. The computers in a highly parallel machine are much closer coordinated than the computers in a distributed system. The computers in a parallel machine share the same clock and memory structure, are typically managed by the same operating system, and typically, but not in general, share the same algorithm. Conversely computers in a distributed system don't share the same clock, don't usually have a common memory structure, and defiantly, aren't running the same operating system nor algorithm.

In some aspects highly parallel machines are similar to distributed systems. However, a highly parallel machine is not a canonical example of a distributed system.

7.3 No Global State

In a distributed system there is an absence of a global state. The state is divided into many smaller units shared by a small number of computers. This distinguishes distributed computing from databases in which a consistent global state is maintained, or at least, in theory a consistent global state could be reached with the aid of a checkpoint and a transaction log.

In a non-distributed system, global state is typically well defined and maintained. Finite state machines are the heart of many non-distributed systems. Finite state machine definitely have a consistent, well-defined state.

State still plays a role in distributed systems. A small partition in the distributed system unite to define a local state. This state is maintained by this partition for a short period of time and disappears when the partition dissolves. Individual components join new partitions, share a very limited state, then dissolve into other partitions.

7.4 No Global Clock

In a distributed system there are as many clocks as there are systems. The clocks are coordinated to keep them somewhat consistent but no one clock has the exact time. Even if the clocks were some what in sync, the individual clocks on each component may run at a different rate or granularity leading to them being out of sync only after one local clock cycle.

Time is only known within a given precision. At frequent intervals, a clock may synchronize with a more trusted clock. However, the clocks are not precisely the same because of time lapses due to transmission and execution.

Consider a group of people going to a meeting. Each person has a watch. Each watch has a similar, but different time. Even with the error in time, the group is able to meet and conduct business. This is how distributed time works.

This is in contrast to a clock on a single system. Here there is only one clock and it provides a unified time for all sub components on this individual system.

7.5 Partial Failures

At any one time, many elements of the distributed system may have failed. If the distributed system is designed correctly, these failures have little visibility to the customer of the system. This property is called high availability and is usually realized by replication of a service over multiple components and by duplication of information.

Many distributed network protocols, like UDP, have as an underlying assumption that there are failures and that packets are lost. By design, the protocol automatically recovers from many classes of failures. This recovery happens nearly transparent to the customer. From the customer's prospective functionality is not lost though response time might be slower.

If the distributed system gradually loses capabilities as more and more of the elements in the system fail, it is said to exhibit graceful degradation. Just because some of the services are not available does not mean that useful work can not be accomplished. A journal or log file might be created to batch the changes when eventually the service becomes available.

Eventually, a point is reached where so many elements have failed that a distributed system will dissolve to individual partitions. The re-grouping of the partitions is usually accomplished with the aid of voting algorithms after quorums have been established.

Many distributed systems application can tolerate one point failure of their underlying services. This happens when any one point in the system can fail, yet the application, as a whole, continues to correctly function. This is called one-point-failure safe. Of course, in a similar fashion, an application can be two-point-failure safe. In general, as the number of tolerated failures increase in an application so does the complexity and cost of the system to support such an application.

An application that never fails during extended periods of time due to hardware errors is fault tolerant. Fault tolerant hardware usually includes triple redundancy for every component with a vote and a compare unit to establish results and to detect potential faults. Each unit has three identical copies each running exactly the same software. The output results are given to a compare unit. If all three units have the same output, the results are considered correct. If two of the three units agree but the third differs, the third unit is considered at fault and the results of the common two are presented as the answer.

7.6 Asynchronous Communication

In a distributed system, communication between elements is inherently asynchronous. There is no global clock nor consistent clock rate. Each computer processes independently of others. Some computers in the system have fast clock cycles while others have slower clock cycles. Even if time was precisely the same on every element in the distributed system, each element would still process the communication at different rates, thus making the commutation asynchronous.

Operating systems have had asynchronous communication for decades. Requests are queued and eventually results return. Asynchronous communications were necessary to manage peripheral devices with a vast spectrum of clock rates. Distributed applications face similar problems. In a traditional computer language the underlying implicit architecture is a call-frame stack. In contrast, the underlying architecture mechanism for asynchronous communication is a queue. There is limited support architectures based on asynchronous communication built from queues in traditional computer languages.

Communications are the dominant aspect of distributed systems. A large part of the architecture description defines the interfaces and communications of the disparate components of the distributed system.

7.7 Distributed Control

In a distributed system there is no central point of control. Rather, a decentralized management paradigm is in place. Each unit, nearly independently, decides to which extent they wish to participate in the distributed system. Individual systems enter and leave the distributed system at will and decide, autonomously, which components are useful.

The distributed system has no one control. Actions of individual systems are governed by policy guidelines where compliance is optional. In general, a guideline change closely reflect the actual behavior change of the distributed system.

There is no one control point for the total distributed system but the distributed system relies on the availability of core underlying infrastructure to make all of this possible. For each core infrastructure service in the distributed system, in contrast, there is a well defined control point. From this control point the core infrastructure services are managed. The collection of these core infrastructure control points enable the existence of the distributed system and empower the remaining systems to exhibit autonomous distributed control.

7.8 Heterogeneous Systems

The distributed system contains many different kinds of hardware and software working together in cooperative fashion to solve problems.

There may be many different representations of data in the system. This might include different representations for integers, byte streams, floating point numbers, and character sets. Most of the data can be marshaled from one system to another without losing significance. Attempts to provide a universal canonical form of information is lagging.

There may be many different instructions sets. An application compiled for one instruction set can not be easily run on a computer with another instruction set unless an instruction set interpreter is provided. There is no universal binary making process migration difficult. Recent developments in the web and Java may provided a universal interpreted language on most computers. Though a computer language is not an instruction set, this is a good compromise.

Some components in the distributed system may have different capabilities than other components. Among these might include faster clock cycles, larger memory capacity, bigger disk farms, printers and other peripherals, and different services. Seldom are any two computers exactly alike.

7.9 Autonomy

There are two distinct categories of autonomy. One category reflects local system autonomy. The other category reflects the autonomy associated with the core infrastructure distributed services that form the heart of a distributed system.

Each local system, in a distributed system, is highly autonomous. They may have entirely different policies and usage from the whole, can decide to which extent they wish to share, and may join and depart, at will, from the distributed system.

However, once a local system decides to enter the distributed system, it is bound to a set of contracts governing behavior, communications, and interfaces. By following the contracts, the local system can then use the services provided or even provide services itself.

The systems that materialize the core infrastructure services have more restrictions on their autonomy. They must have consistent, well-defined communication protocols and follow all contracts. Even though they are managed independently of the local systems, they are bound by a contract to provide a certain level of expected service.

7.10 Evolution

Distributed systems follow an evolutionary development track. Seldom is a distributed system built from scratch with entirely new components. Rather, a construction, in situ, transforms the system from one configuration to another and evolves the system in an incremental fashion.

This evolutionary process of change is one of the distinguishing features of a distributed system. The more traditional approach is to completely replace a non-distributed system with a completely new service. Because of the massive amount of change associated with throwing out the old system and replacing it with a new system this approach is sometimes called a revolutionary approach.

Distributed systems use **E**volution not **RE**volution.

There are two major ways in which evolution can happen in a distributed system. The first way is to keep the interface and the contract constant and replace a service with a different implementation of the service. The new implementation may have better resource usage characteristics thus providing a higher quality of service.

The second way is to keep the old interface and contract in place as well as the old service. A new service, with a backwards compatible interface and contract, is introduced. Over time, the older service is phased out and replaced with the newer service. There may be several different versions of a service active at any one time. Sometimes, older services are never removed.

The distributed system is malleable to change. Its very nature is constant change.

Change is the only constant.

A distributed system has many disparate services. One service can come and go, without interfering with another service. Even though two services may share many common resources, for the most part, they are independent. Of course, if one service relies on another service, then they are cooperative. If the service's partner is not available then the service will not be able to complete its task.

7.11 Opaqueness

Many concepts are hidden from individual systems that make up the distributed system. An individual system might be relocated without customer visibility. There may be many equivalent services. The data might be moved without customer visibility. A service might cover for a failing service.

Complete opaqueness in practice is difficult but in theory, if two services provide the same level of service and support the same contract, then they can be substituted for each other and the customer should not be able to tell.

Not all information should be hidden. If a customer really wants to know which of the many systems is providing the service, that information should be available. Likewise, a customer should be able to specify a particular service from a collection of services.

In general, most users don't care for this level of detail especially if equivalent services can be offered. How often do you get to specify the routing path of a packet? How often do you want to specify the path? It is only the rare exception where this becomes an issue.

7.12 Openness

In this mixture of heterogeneous distributed systems, individual systems seem to cooperate quite well. This is accomplished by the establishment of open (a.k.a. public) protocols, conventions, and standards.

Openness refers to the ability to plug and play. You can, in theory, have two equivalent services that follow the same interface contract, and interchange one with the other.

> Standards are wonderful because there are so many to choose from.

Some standards are created by working groups to unify disparate solutions to problems. Other standards are created, de facto, by companies with the overwhelming market share. In either case, one interface is presented for other application programming interactions.

7.13 Interdependent

Many services across the distributed system may be interdependent. In the sprite of reuse, a service readily will ask another service for help. One service communicates with another service to jointly solve a problem. Past answers may be cached and reused. Default answers may be available on the local host.

Meta data, data, and function sharing are prevail. Services in the distributed system really act like a collection of "virtual" services as they share more and directly implement less. Component assembly is the paradigm for building services.

Interdependence is in conflict with change. The more a service depends on other services the more potential conflicts may arise when a change occurs. Interfaces and contracts in a distributed system must be very static and stable to accommodate both change and interdependence.

It is very simple to extend a service, but much more difficult to shrink or remove a service. In fact, old services are still provided by having the old service evoke a newer service. Many services include undefined extension parameters to accommodate future changes.

Alternatively, two services communicating over an interface could dynamically negotiate the protocol and contract. This implies that the meta interface needs to be static and stable allowing for a more dynamic definition of the service interface.

Telnet is an example of a protocol with interface negotiation while FTP is an example of a protocol without interface negotiation. As a result, you see many versions of FTP running on a system while only one version of TELNET is present.

7.14 Federation

The management of the distributed system resources is federated across many autonomous sites. This includes name spaces where different portions are managed by independent parties yet combine to form one unified name space.

Some services may require a consistent name space. There can not be dangling references. A database is an example. Some service, such as the web, exist quite well in a name space that is incomplete and inconsistent. Both kinds of name spaces can coexist. In fact, portions of the name space can be consistent while other sections may be untrustworthy.

7.15 Security

Security becomes even more important in a distributed system. Authentication, authorization, digital signatures, non-repudiation, encryption, and privacy become major issues as we extend the distributed system. The four basic goals of a security system are to protect information, to detect an intrusion, to confine the security breach, and to repair the damage and return the system to a known stable and secure state.

This proposal does not concentrate on security, but will assume a robust underlying infrastructure that provides these services.

8 Current Distributed Architectures

A distributed system can be composed of a family of basic system architectures.

8.1 Basic Client/Server

See Figure 3 on page 60

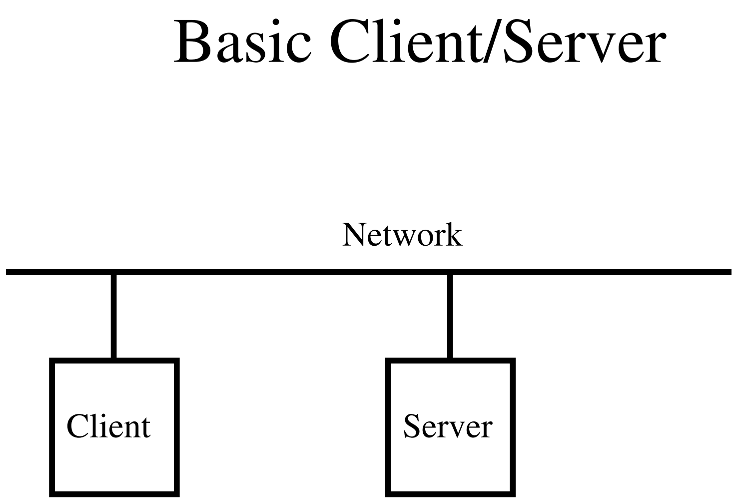


Figure 3: Basic Client/Server

In a client/server architecture the service is divided into at least two pieces. One piece, running on a local user machine, is called the client. The other, running on potentially another machine, is called the server. Traditionally the server provides the computational power or a database engine. The client concentrates on the presentation layer or GUI.

Data is sent over a network from the client to the server. The client and server may be involved in a protocol where commands and meta information are sent along with the data. Since the client and server may be of different hardware architectures marshaling the data is important. Some environments provide secure, encrypted channels and then send all data over these channels while other environments encrypt the message and send the data over the non encrypted channel. Some data may be digitally signed with a MD5 checksum.

The logic in the application may reside at either client or server but traditionally the application logic resides at the client, making the client "fat".

The client server architecture allows for the data to be at one server location while the access of the data to be distributed on many clients. In addition, a larger computer can be used at the server location to meet system response requirements. Many different clients that follow the conventions of the server can reuse this architecture.

Typical client/server services use messaging or remote procedure calls for underlying communication paradigms.

8.1.1 Messaging

Messaging is associated with connectionless, one-way communication between two services. A return message is optional. Two way communication can be supported by using two messages. Messaging is asynchronous by nature. Some systems support persistent messages by placing the messages on a persistent storage queue.

8.1.2 Remote Procedure Call(PRC)

RPC is associated with a connection where two way communication is supported. RPC is synchronous by nature and is an example of request and reply paradigm. RPC is popular with application programmers familiar with the traditional function call paradigm.

8.2 Three Tiered Client/Server

See Figure 4 on page 62

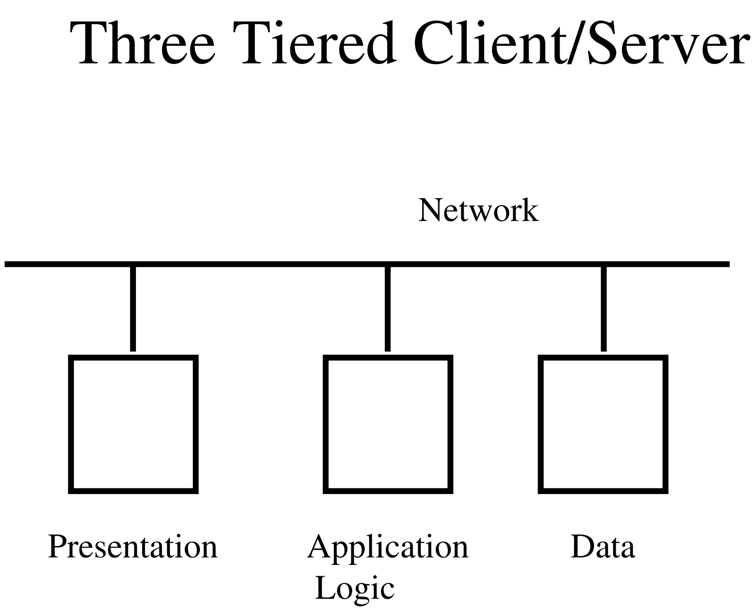


Figure 4: Three Tiered Client/Server

Sometimes an application may be tiered in at least three distinct pieces. One piece is the presentation or graphical user interface(GUI). Another piece is the logic, functions, procedures, and objects that materialize this particular application. Well another piece is the data management most often materialized by a database.

The three tiered architecture allows for one central server location for all the business logic and one central server location for all of the data leading to reuse, consistency, and uniformity of applications in this environment.

The presentation layer supports the GUI. Sometimes this is called a GUI-lite application. There may be many different kinds of GUI.

The middle section materializes the application. Though this is called the application logic it is really much larger then just logic formulas and includes all function and procedures that make the application an application for a particular domain. Many of the elements in this area are tailored for the application needs. Sometimes this layer is called the business rules or logic.

The last layer is the data. This is most often materialized by a database. This layer does not have to be limited to a database. Alternatively, a large computational server could act as a mathematical engine.

The three tiered layering is popular because it forces clean lines between the three layers. The "thin" client can easily be moved to other architectures. The application logic can be used by other applications in the system. The database layer allows for plug and play of many different database vendors.

Each layer may have a special interface or use a more generic interface protocol. An example of a GUI protocol is xterm or html. An example of a database protocol is SQL/Net.

8.3 Five Network Placements

Where is the network interface located in an application? Where the network is placed has significant influence on the architecture. See Figure 5 on page 64.

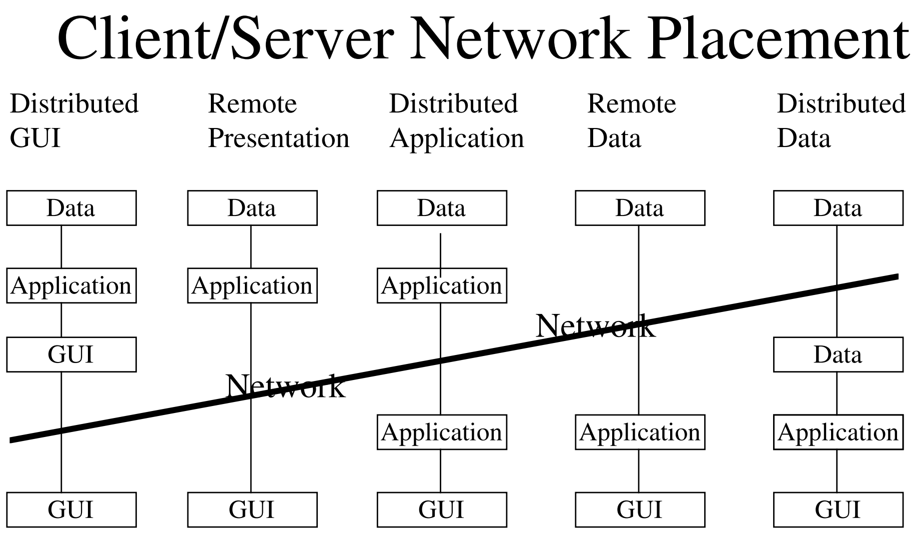


Figure 5: Network Positions

In the traditional three tiered layering of an application, the network is at every layer. This does not have to be the case. We can place the network at many different locations including the middle of a layer.

8.3.1 Distributed Presentation

The presentation layer can be split. The server that does the work for the presentation layer is on one side of the network while the screen is on the other side. A traditional terminal is an example of this layering as well as screen scrapers.

8.3.2 Remote Presentation

The X windowing system is an example of this strategy. The presentation layer talks to the X-Protocol. The location of the screen is independent of the location of the rest of the application. The X-Protocol still needs a computer to materialize the GUI on the screen unlike the previous layering.

8.3.3 Distributed Application Logic

Some of the logic that makes the application resides on the client and some of the logic resides on the server. This is popular with stored procedures in the database technology. The logic that is close to the data is placed in the stored procedures residing near the data. The logic that is more function driven is evoked in the client.

8.3.4 Remote Data

This is sometimes called "fat client". The client has both the presentation and application logic. Only the database component is access over the network. Most of distributed applications written in the early 1990s are based on this technology.

8.3.5 Distributed Data

Here the database is distributed over the network in both vertical and horizontal partitions. The application needs not know where the data is located.

8.4 Publish/Subscribe

In a publish/subscribe architecture, services publish their availability while clients subscribe to services. A client can subscribe to a future service by queuing to the event. In many systems, objects are published including data and the methods to manipulate the data. For efficiency reasons, seldom are individual copies of the methods distributed with every object. Rather, a repository of methods is provided with run time linking and execution support. See Figure 6 on page 66.

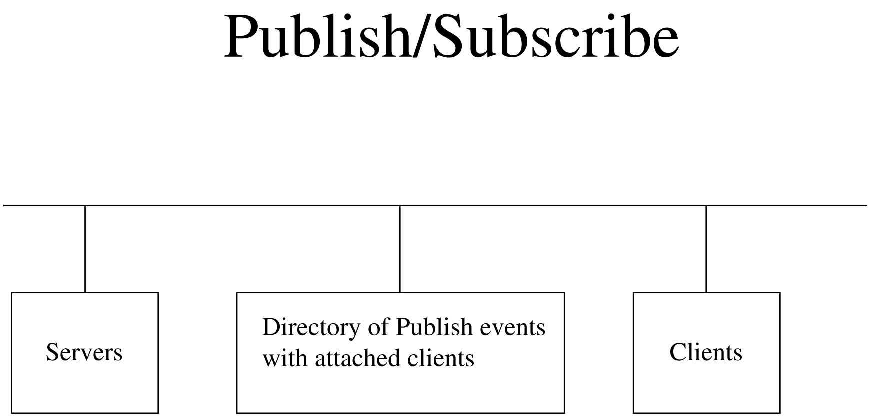


Figure 6: Publish/Subscribe

8.5 Mediator

Consider the problem of many similar, but slightly different, clients wanting to talk to many similar, but slightly different, servers. In the general case you would need N(N-1)! connections. This does not scale very well with large systems.

The mediator sits in the middle. The clients talk to the mediator and the mediator talks to the servers. Only 2N interfaces need to be built and maintained. The mediator can be replicated as needed for performance reasons.

A mediator acts as the glue between many clients and many servers. See Figure 7 on page 67.

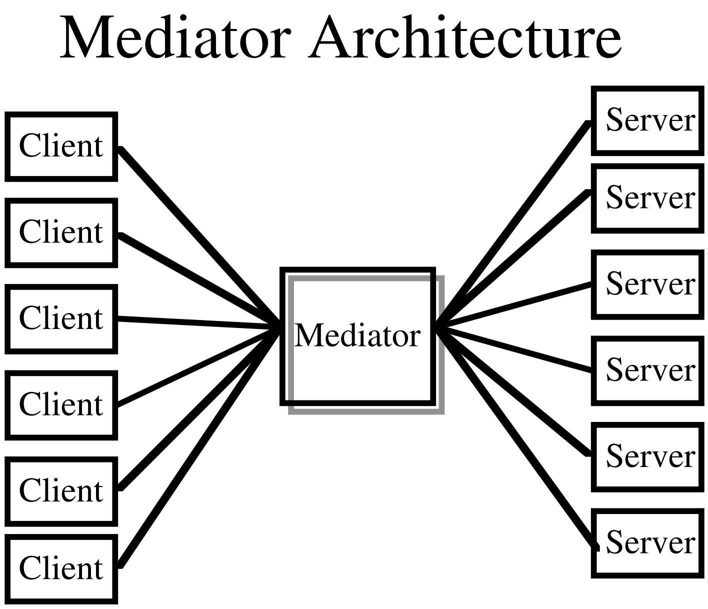


Figure 7: Mediator

8.6 Electronic Mail(email)

One of the most well known distributed applications is email. This application has been around for a couple of decades and materializes millions of email messages per hour world wide. See Figure 8 on page 68

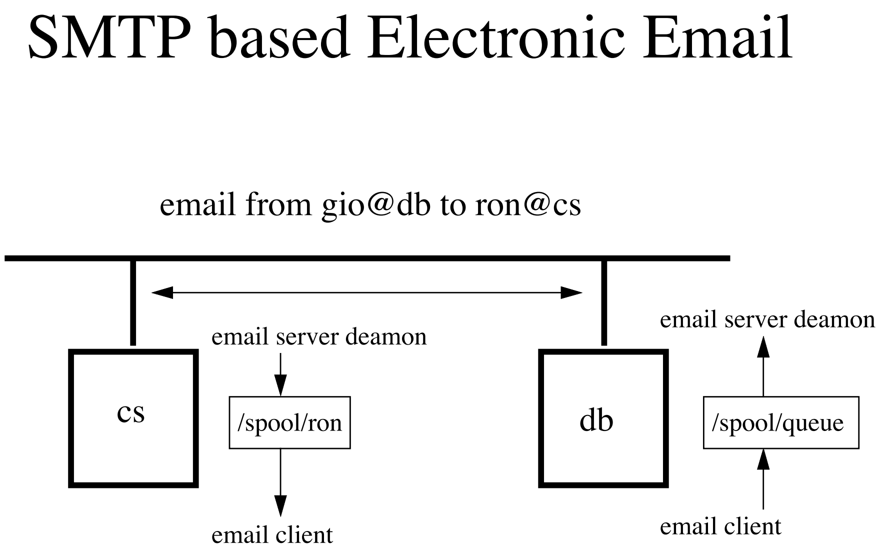


Figure 8: SMTP based email

The email application is the pinnacle of distributed applications. Each computer accomplishes one small step in the process in an autonomous fashion. Individual components may fail while the system remains available to most of the users. The architecture is asynchronous, limited-shared-state, highly available, elaboration tolerance, message passing, with a dynamic protocol.

The big picture of how this works is simple. Each machine has an application server program that understands the email protocol of SMTP. When two machines want to exchange email, first they synchronize to a known initial shared state, then the email message header control information is exchanged. If the second machine agrees to the transfer, the contents of the email is then delivered. The receiving machine places the email in a special spool directory for later reading by the end user. Alternatively, the receiving machine may reject the email message or provide forwarding information.

The next piece of the big picture are the email clients. When the end user wants to send email, a email client hands the email to the machine's email server. The server exchanges the email with another machine's email server. The recipient machine places the email message in the spool file belonging to the recipient. The email is ready for the final delivery stage. When the recipient wants to read email, the email client reads the spool file and displays the message.

If a server is not available the message is queued for later delivery. If an error occurs during transmission, the message is resent. Messages sent to unknown recipients are bounced back to the sender. An email address may include routing information, in which case, the receiver machine accepts the message, removes it's name from the routing list, and then processes the message to the next machine.

Email has survived many major upgrades. MIME extends the simple text message to include other data types. This is accomplished by providing type indicators on the data. When a particular data type appears in a message, and if your email client is MIME smart, the application that understands that data type is evoked and the data is displayed.

The POP protocol groups messages into message boxes. A pop smart email client will acquire all messages in the message box at the same time. The IMAP protocol groups messages into an indexed database. A IMAP smart client can query the index to find the appropriate message.

The email application is not perfect. There is no authentication making the email system autonomous and easily spoofed. There is no data privacy of the message with many opportunities for an outside third party to discover the contents of a message. There is no data integrity of the message allowing for the substitution the message, in transit, without knowledge of the recipient. There is no digital signature on the message that guarantees that the sender really sent this message. There are no guarantees on delivery or performance and no way of specifying different levels of service. In particular, there is no way to verify that a message was delivered and read by the recipient. The recipient could repudiate the receipt of the message.

8.7 The Web

The Web is a new entry into the distributed application arena. See Figure 9 on page 70.

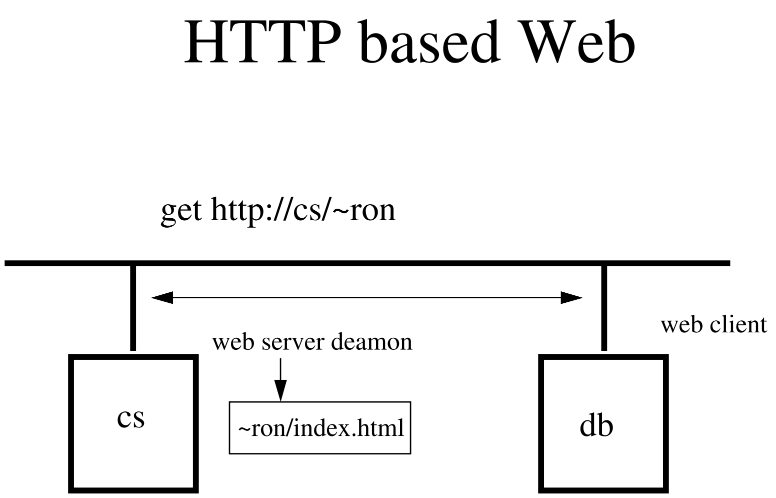


Figure 9: HTTP based Web

The big picture of the Web is similar to email. The system consists of Web servers and Web clients. The server is responsible for materializing pages of html encoded data. The client is responsible for displaying this information. The protocol spoken is called HTTP. The data is handled in a similar fashion as in MIME email.

There are two significant developments that empower the Web. One is the federated name space and the other is stateless computation. The federated name space defines a name for every object on the Web. The Web client can resolve the name and materialize the object. The space is federated allowing for anyone to splice into the name space at virtually any junction. The stateless computation gives incredible flexibility for machines entering and leaving the Web.

The Web is not perfect. As in email, the Web is unauthenticated, non-encrypted, does not have digital signatures, is easily spoofed, with no guarantee of service. The Web has become simultaneously the world's largest source of information and the world's largest source of miss information. There is indication of quality of content or availability of servers. In addition, the name space quickly degrades into stale and out-of-date entries.

9 Related Work

9.1 Programs

9.1.1 DSSA

The DARPA DSSA (Domain Specific Software Architecture) [MG92] program's goals are to develop conventionalized software architectures in various domains, to advance software architecture specification technology, to analyze architectural influences on system performance, and to build tools for software system composition through reuse.

9.1.2 STARS

The DARPA STARS [STA93](Software Technology for Adaptable, Reliable Systems) goals are to increase software productivity, reliability, and quality by integrating support for modern software development process and reuse concepts within software engineering environment technology. STARS uses a technology approach called mega programming paradigm.

9.1.3 CARDS

The Air Force CARDS effort is dedicated to furthering systematic software reuse. This effort has built a library for requirements, architecture and component information as defined in the PRISM Generic Command Center Architecture.

9.1.4 PRISM

The Air Force PRISM program developed an architecture to maximize the reuse of existing COTS and GOTS components in command centers.

9.1.5 DSRS

The DSRS (Defense Software Repository System) contains assets with reuse potential as identified thorough domain analysis. Assets are software development products such as requirements, design specifications, architectures, design diagrams, source code, documentation, test suites, and repository support items.

9.1.6 SATI

The SEI's SATI (Software Architecture Technology Initiative) is to establish a software architecture knowledge repository for the DSSA.

9.1.7 Prototech

The DARPA Prototech program's goal is to enable use of incremental development and prototypeing as a means to increase effectiveness of systems and systematically reduce risk over the software life cycle, focusing particularly on requirements engineering and systems design.

9.1.8 DARPA Software Foundations

The DARPA Software Engineering Foundations Program focuses on technology for developing and supporting high assurance software systems and on technology related to languages used in systems integration. This program includes trusted systems, software understanding, and composition methods.

9.2 ADLs

9.2.1 DICAM

DICAM (Distributed Intelligent Control and Management) ACL was developed by [HRCP94] for the DSSA program. DICAM is a reference architecture for control. Applications are built as hierarchically organized controllers, each with district domain and meta-control components, with a unifying store for world models and shared information. DICAM is not inherently domain specific.

9.2.2 GenVoca LE

GenVoca LE language [BCGS93] is a domain independent model for defining scalable families of hierarchical systems as compositions of reusable components by meta models of large scale system construction.

9.2.3 Capture

(aka KAPTUR - Knowledge Acquisition for Preservation of Trade-offs and Underlying Rationales) Capture [Bai92] embodies a domain analysis approach that combines elements of object-oriented modeling, feature modeling, and case based reasoning.

9.2.4 LILEANNE

LILEANNE [BSTS93] (library Interconnect language extended with annotated Ada) is intended to support abstraction, composition, and reuse of Ada software.

9.2.5 MetaH

MetaH [Ves94a] intended to support analysis, verification, and production of real-time fault tolerant secure multi processing embedded software.

9.2.6 ControlH

ControlH [Ves94b] intended to be used to describe guidance, navigation, and control algorithms in a concise and rigorous manner. ConrolH is tailored for this domain.

9.2.7 Rapide

Rapide [LV96] intended to support the specification, analysis, and verification of system architectures composed of event processing components.

9.2.8 UniCon

UniCon (a language for Universal Connections) [SDK+94] emphasizes the structural aspects of software architecture and is based on the complementary constructs of component and connectors.

9.3 System Environments

9.3.1 UNAS

UNAS (Universal Network Architecture Services) A commercial product by TRW and Rational consisting of a suite of predefined, reusable Ada building blocks, services, and instrumentation that represent the high level primitives for the architecture of a distributed and heterogeneous software system. Provides core executive functions for distributed systems such as initialization, system mode control, reconfirmation, fault detection, health and status monitoring, and inter process communication. High Portability is supported.

9.3.2 DCE

The Distributed Computing Environment (DCE) gives us an interface definition language (IDL), Security, A means of communication using Remote Procedures Calls (RPC), distribution, replication, and many other transparencies.

9.3.3 ODP

See [ODP95d], [ODP95a], [ODP95b], and [ODP95c].

9.3.4 ANSA

ANSA introduces the concepts of binding, trading, and process groups.

9.3.5 CORBA

CORBA introduces the concept of a trader/broker architecture with an IDL and, of course, support for objects.

9.3.6 COM

COM is MicroSoft's version of CORBA.

A Glossary

Abstraction: A description of a family of systems that is independent of the details of any one particular system. [Rei92]

Abstract Syntax: A description of a data structure that is independent of machine-oriented structures and encodings. [JL91]

Abstract Syntax Notation One (ASN.1): The language used by the OSI protocols for describing abstract syntax. This language is also used to encode SNMP packets. ASN.1 is defined in ISO documents 8824.2 and 8825.2. [MP93]

Access Control List (ACL): Most network security systems operate by allowing selective use of services. An Access Control List is the usual means by which access to, and denial of, services is controlled. It is simply a list of the services available, each with a list of the hosts permitted to use the service. [MP93]

Advanced Research Projects Agency Network (ARPANET): A pioneering long haul network funded by ARPA (now DARPA). It served as the basis for early networking research, as well as a central backbone during the development of the Internet. The ARPANET consisted of individual packet switching computers interconnected by leased lines.

Agent: In the client-server model, the part of the system that performs information preparation and exchange on behalf of a client or server application. [JL91]

American National Standards Institute (ANSI): This organization is responsible for approving U.S. standards in many areas, including computers and communications. Standards approved by this organization are often called ANSI standards. ANSI is a member of ISO. [MP93]

American Standard Code for Information Interchange (ASCII): A standard character-to-number encoding widely used in the computer industry. [MP93]

Application: A program that performs a function directly for a user. [MP93]

Application Program Interface (API): A set of calling conventions which define how a service is invoked through a software package. [JL91]

Architecture: The components of a system and their interface behavior.

Atomicity: An action either happens or it does not in total, there is no partial completion.

Authentication: The verification of the identity of a person or process. [MP93]

Bandwidth: The amount of data that can be sent through a given communications circuit. [MP93]

Behavior: The responses of the components in a system to each other and other stimulation from the environment.

Big-endian: A format for storage or transmission of binary data in which the most significant bit (or byte) comes first. The term comes from "Gulliver's Travels" by Jonathan Swift. The Lilliputians, being very small, had correspondingly small political problems. The Big-Endian and Little-Endian parties debated over whether soft-boiled eggs should be opened at the big end or the little end. [JL91]

Binding: The mapping of an abstract name to more detailed representation. The IP address of a computer given the name of the computer is an example of a binding. The port of an application is another example.

Channel: A communication path that guarantees delivery and correct order of data.

Checkpoint: A particular point in a computation where state is written to stable storage. At some later time, the computation may be initialized to this saved state.

Class: The meta structure of an object defining the inheritance of methods and data structures.

Client: A computer system or process that requests a service of another computer system or process. A workstation requesting the contents of a file from a file server is a client of the file server. [MP93]

Communication: An exchange of information.

Compliance: An implementation is in compliance with a specification when all conditions and expectations hold.

Composition: A technique used to combine smaller components into large components to solve more complex problems.

Connection-oriented: The data communication method in which communication proceeds through three well-defined phases: connection establishment, data transfer, connection release. TCP is a connection-oriented protocol. [MP93]

Connectionless: The data communication method in which communication occurs between hosts with no previous setup. Packets between two hosts may take different routes, as each is independent of the other. UDP is a connectionless protocol. [MP93]

Contract: An agreement between two components in a system determining their behavior.

Data: Bases level measurement of uninterrupted information. Example could include integers, floating point numbers, and dates. Data is consumed by analysis techniques which generate information.

Decomposition: A technique used in modeling where more complex systems are separated into smaller less complex systems. This process continues until a system is reached where a know solution exists.

Distributed Computing Environment (DCE): An architecture of standard programming interfaces, conventions, and server functionalities (e.g., naming, distributed file system, remote procedure call) for distributing applications transparently across networks of heterogeneous computers. Promoted and controlled by the Open Software Foundation (OSF), a consortium led by Digital, IBM and Hewlett Packard. [JL91]

Domain Name System (DNS): The DNS is a general purpose distributed, replicated, data query service. The principal use is the lookup of host IP addresses based on host names. The style of host names now used in the Internet is called "domain name", because they are the style of names used to look up anything in the DNS. [MP93]

Encapsulation: The technique used by layered protocols in which a layer adds header information to the protocol data from the layer above. [JL91]

Encryption: Encryption is the manipulation of data in order to prevent any but the intended recipient from reading that data. There are many types of data encryption, and they are the basis of security. [MP93]

Environment: The underlying system assumptions, resources, contracts, and behaviors as viewed by a component.

Error: A state in computation which leads to unexpected conditions. This is also called a failure or a fault.

Extended Binary Coded Decimal Interchange Code (EBCDIC): A standard character-to-number encoding used primarily by IBM computer systems. [MP93]

Federation: An organizational technique of a system where many decisions are decentralized and distributed, yet individual members benefit from each others contributions.

Information: Low content data is analyzed generating higher content information.

Interface: The layer between two components in a system.

Invariant: A quantity that does not change during a computation.

International Organization for Standardization (ISO): A voluntary, non-treaty organization founded in 1946 which is responsible for creating international standards in many areas, including computers and communications. Its members are the national standards organizations of the 89 member countries, including ANSI for the U.S. [MP93]

Kerberos: Kerberos is the security system of MIT's Project Athena. It is based on symmetric key cryptography. See also: encryption. [MP93]

Little-endian: A format for storage or transmission of binary data in which the least significant byte (bit) comes first. See also: big-endian. [JL91]

Migration: The moving of one process in a system from one physical location to another.

Name: An identifier, most often unique, to a component in a system.

Name Resolution: The process of using the name to obtain more information.

Name Space: A collection of unique names.

Notification: Awareness of a state change regardless of location.

Object: A component of a system that logically contains both data and methods.

Permission: In security terms, this is the granting of authorization.

Persistence: An component that survives for a long period of time usually associated with data stored on a file system.

Policy: A high-level guideline governing behavior.

Port: A port is a transport layer de-multiplexing value. With Each application is a unique port number. Network packets are first routed to the computer using the IP and then routed to the application using the port number. [MP93]

Prohibition: An action that is not permitted to happen.

Protocol: A formal description of message formats and the rules two computers must follow to exchange those messages. Protocols can describe low-level details of machine-to-machine interfaces (e.g., the order in which bits and bytes are sent across a wire) or high-level exchanges between allocation programs (e.g., the way in which two programs transfer a file across the Internet). [MP93]

Quality of Service (QoS): A collection of measurements indicating how well a system is behaving. These might include response time, bandwidth, latency, error rates, recovery time, down time, round-trip-time, and many more.

Refinement: An analysis process which takes one level of a design into a more detailed level.

Remote Procedure Call (RPC): An easy and popular paradigm for implementing the client-server model of distributed computing. In general, a request is sent to a remote system to execute a designated procedure, using arguments supplied, and the result returned to the caller. There are many variations and subtleties in various implementations, resulting in a variety of different RPC protocols. [JL91]

Request For Comments (RFC): The document series, begun in 1969, which describes the Internet suite of protocols and related experiments. Not all (in fact very few) RFCs describe Internet standards, but all Internet standards are written up as RFCs. The RFC series of documents is unusual in that the proposed protocols are forwarded by the Internet research and development community, acting on their own behalf, as opposed to the formally reviewed and standardized protocols that are promoted by organizations such as CCITT and ANSI. [MP93]

Security Authority: One component of a system that provides the services of authentication, authorization, integrity, confidentiality, and non-repudiation. This is accomplished using audit trails, encryption, and private and public keys. The four basic behaviors of a security system is to protect, detect, confine, and to mitigate.

Server: The component in a distributed system that provides services.

Signal: A state change that is handled out-of-bounds usually using the mechanism of an interrupt.

State: The items that define a component at a particular time.

Stream: A type of transport service that allows its client to send data in a continuous stream. The transport service will guarantee that all data will be delivered to the other end in the same order as sent and without duplicates. [MP93]

System: The sum total of all the components, their interfaces and behavior.

Thread: An execution path of computer instructions usually at the control of an application sharing one address space.

Transparency: A component exhibits transparency to a change, if no measurable difference exists before and after the change. Examples of transparency might be different locations, different operating systems, or different protocols.

Type: The class of which the object belongs.

B Acronym Key

ACID	Atomicity Consistency Isolation Durability
ACK	Acknowledgment
ADL	Architecture Definition Language
AD	Applications Development
API	Applications Programming Interface
ARPA	Advanced Research Projects Agency
BSD	Berkeley Software Distribution
BU	Business Unit
CICS	Customer Information Control System
CNOS	Corporate Network Operating System
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
C/S	Client/Server
CSA	Consolidated Security Administration
DARPA	Defense Advanced Research Projects Agency
DBMS	Database Management System
DCE	Distributed Computing Environment
DDCS	Distributed Database Connectivity Services
DLL	Dynamic Link Library
DNS	Domain Name Service
DSM	Distributed Security Manager
DSOM	Distributed System Object Model
DSSA	Domain Specific Software Architecture

EDA/SQL	Enterprise Data Access/SQL
EDI	Electronic Data Interchange
ESA	Enterprise Systems Architecture
EBCDIC	Extended Binary Coded Decimal Interchange Code
FTE	Full-Time Equivalent
3GL	Third-Generation Language
4GL	Fourth-Generation Language
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
IPX	Internetwork Packet Exchange
IS	Information Systems
ISO	International Organization for Standardization
ISV	Independent Software Vendor
IT	Information Technology
IEEE	Institute of Electrical and Electronics Engineers

LAN	Local-Area Network
LOC	Lines of Code
MAA	Microsoft Application Architecture
MIPS	Millions of Instructions Per Second
MOM	Message-Oriented Middleware
MQ	Message Queuing
NAK	Negative Acknowledgment
NDS	NetWare Directory Service
NFS	Network File System
NOS	Network Operating System
NSF	National Science Foundation
OLE	Object Linking and Embedding
OLE DB	OLE Database
OMG	Object Management Group
OO	Object-Oriented
ORB	Object Request Broker
OSI	Open Systems Interconnection
OSF	Open Software Foundation
OS	Operating System
PC	Personal computer
QoS	Quality of Service

RAAD	Rapid Architecture for Applications Development
R&D	Research and Development
RDBMS	Relational Database Management System
RFC	Request For Comments
RPC	Remote Procedure Call
SAA	System Applications Architecture
SNA	Systems Network Architecture
SOM	System Object Model
SQL	Structured Query Language
SSO	Single Sign-On
TCP/IP	Transmission Control Protocol/Internet Protocol
TP	Transaction Processing
URL	Uniform Resource Locator
VB	Visual Basic

References

[AG94] R. Allen and D. Garlan. Beyond definition/use: Architectural interconnection. *Proceedings, Workshop on Interface Definition Language*, January 1994.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[Bai92] Cidnet Bailin. Kaptur - knowledge acquisition for preservation of trade-offs and underlying rationales. *private document, copyright Bailin*, 1992.

[BCGS93] Don Batory, Lou Coglianses, Mark Goodwin, and Steve Shafer. Creating reference architectures: An example from avionics. *anonymous ftp to cs.utexas.edu*, 1993.

[BG77] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058, 1977.

[BG80] R. M. Burstall and J. A. Goguen. The semantics of clear, a specification language. *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332, 1980.

[Bro87] B. W. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[BS92] B. W. Boehm and W. L. Scherlis. Megaprogramming. *Software Technology Conference*, pages 63–82, April 1992.

[BSTS93] Don Batory, Vivek Singhal, Jeff Thomas, and Marty Sirkin. Scalable software libraries. *Proceedings of the ACM, SIGSOFT Conference*, December 1993.

[Coo79] L. W. Cooprider. *The Representation of Families of Software Systems.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, April 1979.

[COR96] *Common Object Request Broaker Architecture, The Object Management Group*, 1996.

[DCE96] *The Distributed Computing Environment, Open Software Foundation, The Open Group*, 1996.

[DEF95a] *ITU-T X.902 — ISO/IEC 10746-2*, 1995.

[DEF95b] *ITU-T X.903 — ISO/IEC 10746-3*, 1995.

[DK76] F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2(80):80–86, June 1976.

[EHL+94] S. Edwards, W. Heym, T. Long, M. Sitarman, and B. Weide. Specifying components in resolve. *Software Engineering Notes*, 19(4), October 1994.

[GA84] Goldberg and Adele. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984. ISBN 0-201-11372-4.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, April 1995.

[GAR83] Goldberg, Adele, and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983. ISBN 0-201-11371-6.

[GAR89] Goldberg, Adele, and David Robson. *Smalltalk-80: The Language.* Addison-Wesley, 1989. ISBN 0-201-13688-0.

[GP95] David Garlan and Dewayne Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, April 1995.

[HKM+94] Nicholas Haines, Darrell Kindred, J.G. Morrisett, Scott M. Nettles, and Jeanette M. Wing. Composing first class transactions, a functional standard metalanguage packages of concurrrency, locking, persistence, and serialization. *ACM TOPLAS*, 16(6):1719–1736, November 1994.

[HL85] D. Helmbold and D. Luckham. Tsl: Task sequencing language. *The Ada International Conference, Paris, France*, May 1985.

[HLN+88] Harel, Lachover, Naamad, Pnueli, Politi, Sherman, and Shtul-Trauring. Statemate: a working environment for the development of complex reactive systems. *Proceedings of the 10th International Conference on Software Engineering, Singapore*, April 1988.

[Hoa94] C.A.R. Hoare. Mathematical models for computing. *Science*, August 1994.

[HRCP94] Terry Hayes-Roth, Erman Coleman, and Devito Papanagopoulos. Overview of technowledge's dssa program. *ACM SIGSOFT Software Engineering Notes*, October 1994.

[JL91] O. Jacobsen and D. Lynch. *A Glossary of Networking Terms, RFC 1208, Interop, Inc.*, 1991.

[JM94] F. Jahanian and A. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, December 1994.

[KG83] Krasner and Glenn. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983. ISBN 0-201-11669-3.

[KO62] Nygaard K. and Dahl O.J. *The Development of the SIMULA Languages*. Norwegian Computing Centre (NCC) in Oslo, 1962.

[LKA+93] Luchham, Kenney, Augusting, Vera, Bryan, and Mann. Specification and analysis of system architecture using rapide. *Stanford University Technical Report*, 1993.

[LSBH93] Luqi, Shing, Barnes, and Hudhes. Prototypeing hard real-time ada systems in a classroom environment. *Proceedings of the Seventh Annual ADA Software Engineering Education and Training (ASEET), Monterey*, January 1993.

[Luc90] D. C. Luckham. Programming with specifications: An introduction to anna, a language for specifying ada programs. *Texts and Monographs in Computer Science*, October 1990.

[LV96] Luchham and Vera. An event-based architecture definition language. *to appear in IEEE Transactions on Software Engineering*, 1996.

[LvH85] D. C. Luckham and F. W. von Henke. An overview of anna, a specification language for ada. *IEEE Software*, 2(2):9–23, March 1985.

[Mat86] R. F. Mathis. The last 10 percent. *IEEE Transactions on Software Engineering*, 12(6):705–712, June 1986.

[Mey92] B. Meyer. *Eiffel: The Language*. Prentice Hall, New York, 1992.

[MG92] Mettala and Graham. The domain-specific software architecture program. *Special Report CMU/SEI-92-SR-9, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA*, June 1992.

[MP93] Gary Scott Malkin and Tracy LaQuey Parker. *Internet Users Glossary, RFC 1392, Interop, Inc.*, 1993.

[NB92] Newton and Browne. The code 2.0 graphical parallel programming language. *Proceedings, ACM International Conference on Super Computing*, July 1992.

[ODP95a] *ITU-T Rec. X.902 — ISO/IEC 10746-2: Foundations*, 1995.

[ODP95b] *ITU-T Rec. X.903 — ISO/IEC 10746-3: Architecture*, 1995.

[ODP95c] *ITU-T Rec. X.904 — ISO 10746-4: Architectural semantics*, 1995.

[ODP95d] *ITU-T X.901 — ISO/IEC 10746-1 Overview*, 1995.

[Ous94] J. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley, 1994.

[PXL95] Palsberg, Xiao, and Lieberherr. Efficient implementation of adaptive software (summary of demeter theory). *Northeastern University, Boxton*, 10, January 1995.

[Rei92] Aaron Reizes. *The Webster Dictionary.* 1992.

[SDK+94] Shaw, Deline, Klein, Ross, Young, and Selesnik. Abstraction for software architectures and tools to support them. *Carnegie Mellon University*, February 1994.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, New York, April 1996.

[STA93] Stars (software technology for adaptable, reliable systems). conceptual framework for reuse processes (cfrp), volume 1: Definition, version 3.0. *STARS-VC-A018/001/00*, October 1993.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[Ves94a] S. Vestal. Mode changes in a real-time architecture description language. *Proceedings, Proc. International Workshop on Configurable Distributed Systems: Honeywell Technology Center and the University of Maryland*, 1994.

[Ves94b] Steve Vestal. A cursory overview and comparison of four architecture description languages. *Honeywell Technology Center, Minneapolis, MN, anonymous ftp to src.honeywell.com in /pub/ARCHIVE/dssa/papers*, July 1994.

[Wie92a] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, March 1992.

[Wie92b] G. Wiederhold. Model-free optimization. In *Proceedings of DARPA Software Technology Conference*, pages 82–96, Los Angeles, CA., April 1992. Meridien Corp., Arlington, VA.

[Wol85] A. L. Wolf. *Language and Tool Support for Precise Interface Control.* PhD thesis, Computer and Information Science Department, University of Massachusetts, Amherst, Mass., 1985.

[WWC92] G. Wiederhold, P. Wegner, and S. Ceri. Towards megaprogramming. *Communications of the ACM*, 35(11):89–99, 1992.