

vaakya™

Distributed Computing Architecture

For

Dynamic / Heterogeneous / Distributed Software
Applications

Version 1.0

Preface

This White paper provides:

- An Overview of “Vaakya Distributed Computing Architecture”
- Components in Vaakya
- Features and benefits

Reference Documents:

- Vaakya Business Application Framework

The reference documents are available for reading and download at www.vaakya.com

Table of Contents

Introduction	4
Evolution of Architectures.....	5
Monolithic Architectures	5
Thick-Client Architectures.....	6
Server-Centric Architectures	6
Evolution of Distributed Computing.....	6
The Vaakya Architecture	8
Concept.....	8
Vaakya Architecture Model	9
Architecture Details	10
Framework Specific Components	10
Infrastructure Components.....	13
Application Development.....	18
Application Deployment	18
Application Execution	19
Summary.....	20
Terminology	21
Abbreviations	23
Contact Us	24

Introduction

Technology and businesses drive, adopt and influence each other and have evolved over the years. The evolution in Information Technology and Software in particular, has been occurring simultaneously on multiple dimensions like eliciting requirements, converting requirements to application execution models, sharing information and more importantly the way applications are accessed.

Rapid advancement in computing devices, data storage and communication capabilities are changing the way people collaborate and conduct business, thereby changing fundamental business models.

Present day business executives often use more than one device to communicate and businesses have moved from document-centric transactions to secured electronic transactions. The all-important currency, voice and video, is electronically captured and stored.

'**Change**', an inevitable element in business places an enormous need for mobile application access, multi-format data storage, secure transactions and adaptive applications.

Highly interactive & productive rich client environments will aid distributed computing, static page content will be replaced with executable programs delivered over the internet, extended network of internet-enabled devices, endpoints & applications will create an environment to track, monitor & control.

Software as a Service subscription, social computing networks monetized by advertisements & other on-demand business models are changing the way applications & services are built & delivered over the web.

Vaakya, a comprehensive "**distributed computing architecture**", built ground up, addresses the above and futuristic needs, to bridge the gap between technology and business.

The Vaakya Architecture and its foundation of components provide the platform to build futuristic domain specific frameworks for business applications, handheld applications, image processing, systems programming or embedded applications.

The "Vaakya Business Application Framework", is the first framework product built on the Vaakya Distributed Computing Architecture, which significantly reduces the "gap between development and deployment environments" of enterprise business software.

Objective

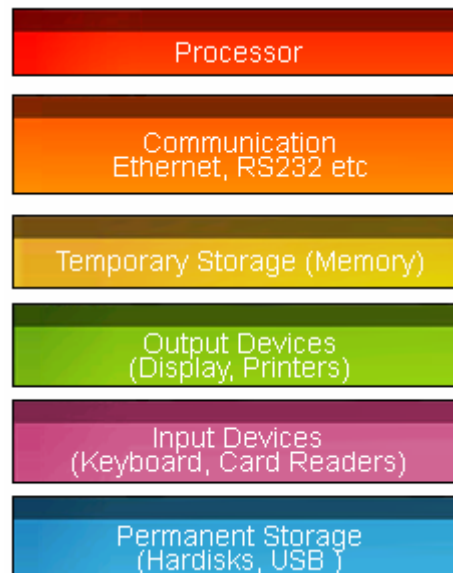
It is our endeavor to provide a compact yet powerful technology platform, using model-driven frameworks and a simple descriptive language, to build and deploy robust applications with relative ease.

Evolution of Architectures

A distributed computing environment would consist of multiple devices with different capabilities and connected to each other over different mediums. Let us look into an evolution that started with a monolithic environment to the current distributed environment.

Monolithic Architectures

Monolithic architectures were built for a specific hardware device. A typical device is expected to have the following hardware components.



These hardware components work to provide a specific functionality. *Operating system* virtualizes the device by providing a single interface to primitive services like file storage, virtual memory management, communication socket and threading, synchronization and device interface services.

The application *executables* developed for these "monolithic" application architectures (as in the case of mainframe) enabled multiple users to share a single device (virtual) and centralize application management.

The main focus at this stage was on "*application execution*" and "*management*" aspects.

Thick-Client Architectures

The advent of desktop computing devices and low cost servers, initially, extended and used the same monolithic application execution model (thick client model).

GUI based desktop development tools along with database servers simplified application development. Thick client executables and database server enabled client-server 2-tier deployment models.

However, in a 2-tier thick-client environment, business rules were spread across both, the client and the database server, which induced difficulty in managing and scaling applications.

Server-Centric Architectures

Evolution of document browsers, middleware like web-servers, application servers and “virtual machines” enabled cross-platform deployment capability and centralized management of applications, leading to “thin client and server centric” computing architectures.

Simultaneously, application development methodologies shifted from structured to object-oriented programming systems. Scripting languages become predominant, co-existing with ‘whole-compilation’ languages, each having their own pros and cons.

Application delivery models improved “server-centric” computing significantly with “plug-in” architectures and “online” updates.

Increased use of applications in business created widespread use of servers and desktops working on disparate operating systems and varied storage capacities. Clusters, Storage Area Networks and similar middleware evolved to virtualize devices.

Modeling tools augmented application development and adoption of XML improved application (data) integration.

The main focus at this stage was in application “*delivery*” and “*management*”.

Evolution of Distributed Computing

Penetration of internet, communication infrastructure and mobile devices has brought about a change in the way applications are developed, deployed and used.

Software business models have transformed from ‘on-premise’ product licenses to “On-Demand” pay per use subscription models using Software as a Service (SaaS).

Mainframes, desktops, servers and mobile devices co-exist to create a distributed environment, but increase complexities for infrastructure components, platforms and tools to cover all aspects of an application lifecycle, including “*collaborative*” development, execution, management, integration, delivery and most importantly *application access*.

Most “application architectures” of today revolve around “server-centric”, thin-client internet browser models. *Browsers, which are primarily based on “Document Object Model”,* have been *arguably* used to create simulated, rich-client application environments.

A shift from middleware and server-centric processing to a shared processing approach is necessary, and, devices in a distributed computing environment should be capable of sharing and processing tasks based on available resources.

To create a truly distributed computing environment with pervasive intelligence, a move beyond browser based thin-client, server-centric computing model is imperative.

An ideal distributed computing environment should contain the following characteristics:

Development

- Descriptive scripting
- Collaborative and iterative development
- Online or offline access

Application access

- Standalone on a PC, handheld or laptop
- A client-server application (on-premise)
- An internet application (web based)
- A hybrid of online and offline

Management and Delivery

- Upgrades to clients and servers
- Fault tolerance and load balancing
- Project Management & metering

Integration

- Message-Oriented
- Application Programming Interface (API)
- Plug-ins

Execution

- Optimal use of memory and processing capability of devices within a distributed environment
- Minimal network bandwidth
- Rich-client user interface

Security

- Identity management
- Secure communication independent of underlying communication protocols

The Vaakya Architecture

Concept

The “**Vaakya Distributed Computing Architecture**” takes a comprehensive approach to address the demanding needs in design, development, deployment and maintenance aspects of distributed software applications.

The philosophy of Vaakya Architecture is, “***to build most of the infrastructure and middleware components into a single runtime and distribute the runtime across all devices***”, thereby virtualizing the complete environment and enabling clients and servers to work *independently or collaboratively*.

The challenges in designing such a single runtime are,

- abstracting and layering of appropriate infrastructure components
- maintaining minimal size of runtime
- distribution and management of runtime

The design strategy of Vaakya architecture is to provide a combination of built-in “***cross-platform infrastructure components within the runtime***” and a blueprint “***to create framework specific micro-components***”.

The *infrastructure components* along with *framework specific micro-components* provide a complete distributed computing environment, to develop and deploy dynamic, distributed and heterogeneous software applications.

Vaakya Architecture Model

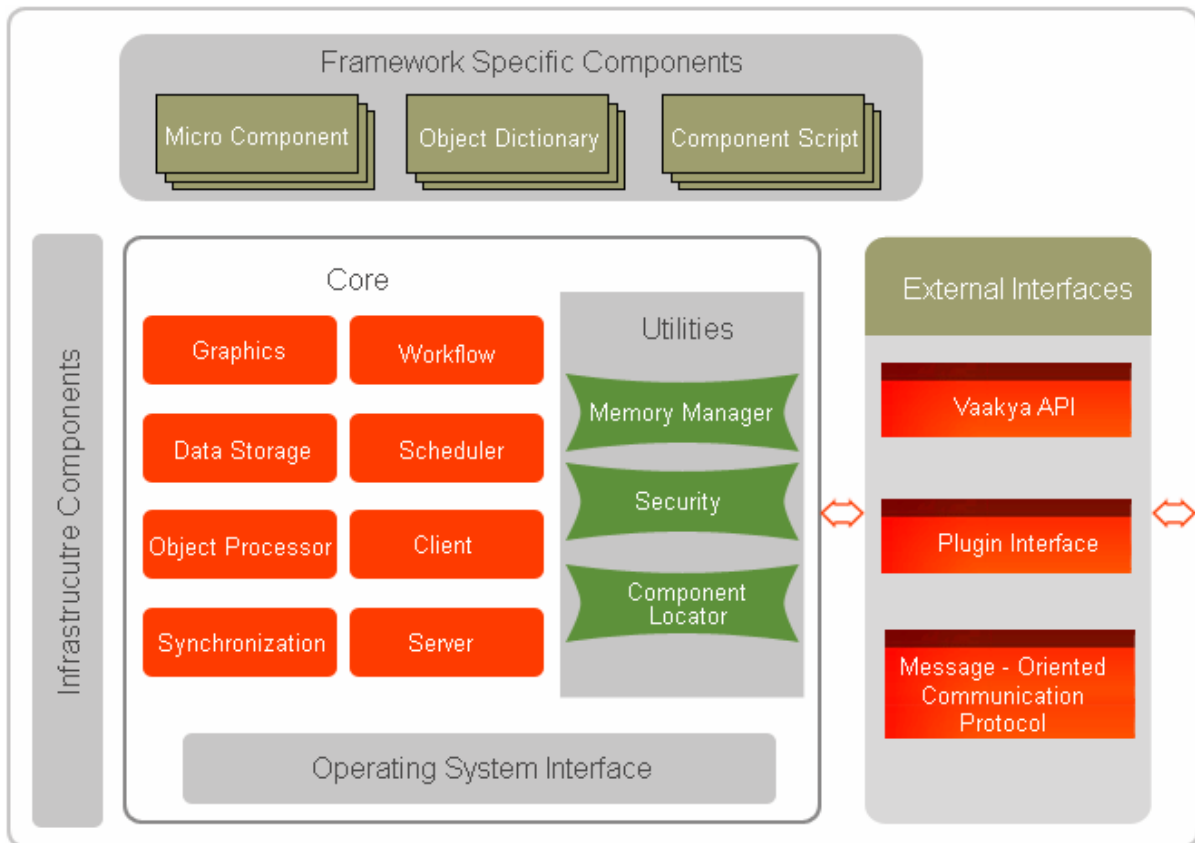
The architecture model below shows infrastructure components and framework specific components, built over an operating system interface, to provide cross-platform deployment capability.

The infrastructure components can be categorized into

- Core components
- Utilities and
- External interfaces

The Framework Specific Components are

- Domain based Functional Micro-Components
- Vaakya Descriptive Scripting Language (Component Specific)
- Distributed Object Dictionary



Architecture Details

Framework Specific Components

Frameworks for a domain are divided into multiple micro-components, an object dictionary and a scripting language to use the micro-components.

Functional Micro-Components

Functional requirements for a framework are built as micro components over cross-platform infrastructure components

Micro-Component Architecture

- Predefined memory structure
- Predefined method implementation
- Vaakya descriptive script
- Script Loader
- Component executor
- Communication interface

Examples of micro-components in a business application framework include Form, Report, User defined functions etc.

Let us consider an example of a Form component to understand the micro component architecture.

A Form component is used for data entry and manipulation. Basically they contain one or more objects, fields and validations.

These patterns of functionality are taken as pre-defined structures. Vaakya script is defined around these structures.

Invocations of these components are based on message requests implemented through the Vaakya communication protocol. All components built should provide these interfaces.

The processor engine will hand over the script to the component loader that is responsible for transferring the contents of the script into the micro-component structure. The loader establishes links to the objects and fields to the object dictionary.

The component executor with built-in functionality for transaction and concurrency management will offer the basic services for addition, modification and deletion of the contents in the form. Memory management is handled by the micro-component.

Dynamic validations and expression evaluations is done by the "Object processor engine".

Script for these components can be made available locally or can be picked up from a resource on the network. This facilitates for centralized application maintenance.

Advantages

- Micro-components can be added or modified as required
- Effective and efficient memory management
- Localized changes do not affect other components
- Service oriented invocation due to VML (Vaakya Markup Language) based Vaakya Communication Protocol
- Most of the component functionality is available in the form of native code to provide maximum performance
- Choice of online, offline or hybrid deployment.

Application developers need not design the application for specific deployment environments.

Framework-based Descriptive Scripting Language

Design, development, testing and change management is considerably made easier by following a framework-based approach.

The **higher-level** Vaakya descriptive scripting language offers simple and intuitive constructs rather than the algorithmic approach.

Advantages

- The descriptive language constructs are built over native executable tested code and hence leads to better performance.
- Changing component behavior in runtime is made easier as it takes just replacement of text in memory, than recompile and create an executable or byte code.
- Very few lines of code are required to achieve a complex set of functionalities.
- Maximum component reuse.
- Text based and non-executable script minimizes intrusion attacks.

Distributed Object Dictionary

Distributed Object Dictionary (held in the RAM) is one of the core elements of "Vaakya distributed computing" architecture.

The meaning of "Object" is context sensitive and was rightly defined as "I Unknown". The term "Object" is used interchangeably to represent real world entities, functions, frameworks or even applications.

Lets us try to understand the context of object and object dictionary with the help of a business application as example.

Consider a software application that can store, fetch and manipulate an employee object for n-tier deployment.

Employee

- Properties
 - Id, name, dept and designation.
- Methods
 - Add, Modify, Delete
- Components
 - User interface, Business rules/relationships and Data access

In Vaakya client-server environment, the property and relationship information is sent across to all the clients that use this application on one time basis or based on rules. The combination of property and relationship information that resides in memory is termed as "object dictionary" in Vaakya environment.

Advantages

- Memory allocation for data storage is done in advance and hence, various components that use these objects need not allocate and de-allocate every time.
- User interface components in the client can pick up the changes to the object directly.
- Server side components can manage the transaction by just knowing the name of the object. The DDL (Data Definition Language), DML (Data Manipulation Language) generation is based on the Object definition and hence can be interoperable with any other DBMS.
- Messages that are transferred across to other systems need not send the property and relationship information again and again. This leads to effective network resource management as well as reduced parsing time (**Appendix - VML construct**).
- **Since the object definition and components are available in each system, the execution environment can be dynamically changed to behave as a client or server.**
- A single application source can work on different type of machines, be it **32 or 64 bit**.
- Enables **on-line, offline or hybrid** application configuration.
- All application specific utilities like backup, recovery, data synchronization etc are "**Object aware**" (Please refer to business application framework document for details of these utilities).
- Maximum reuse in terms of code and memory.

What constitutes an object dictionary, depends upon the application domain like business applications, image processing, systems management or embedded applications.

The distributed object dictionary based approach takes model driven environments a huge step forward in the information technology domain.

Infrastructure Components

The cross-platform infrastructure components are categorized into core, utility and external interface components required for a distributed computing environment.

Data Storage

Vaakya data store is a **file based, record level data storage mechanism** with multiple indexing facilities. The basic functionality offered by the data store is to add, modify or delete one record or a set of records.

The data store also comes with a built-in compacting mechanism used for re-indexing and reorganization of the data store.

Vaakya descriptive scripting language along with **Vaakya Query Language** interface provides a very simple and powerful mechanism for data manipulation.

Business continuity utilities like **backup, recovery, synchronization** etc follow an “object aware” approach and are provided as a part of the framework. This allows for maintaining applications across any of the available DBMS (Data base management systems).

Graphics Engine

The Vaakya Graphics Engine (VGE) is a complete user interface management system and an integral part of the Vaakya architecture. Vaakya graphics engine is built directly on a **cross-platform frame buffer layer**.

The core of the VGE consists of utilities for drawing lines, circles, polygon filling, font handling and graphics related utilities. **Window management, widgets and event handling** (keyboard, mouse and other system messages) are built using the core graphics utilities.

Basic widgets like input box, check box, input grid, tree etc are provided and custom widgets can be created using Vaakya Graphics API.

Event router component in the Vaakya graphics engine takes care of the event handling mechanism for application, window and widgets.

Device Interfaces

Configurable device interfaces are provided to connect various devices like printers, keyboard etc. This allows for deployment time configuration of device properties.

Server

Server component provides the mechanism for accessing Vaakya applications in static, dynamic, synchronous and asynchronous modes. The server component along with the Vaakya communication protocol provides configurable deployment capabilities.

The application server, workflow server, backup server, etc, part of “Vaakya Business Application Framework” are created from this server component.

Vaakya Application Server (VAS) is required for accessing applications in a client-server mode. In the context of business applications, application server takes care of loading and distributing Vaakya components, transaction management, concurrent user access and Column (property in OO terms) level data encryption. **VAS provides a lightweight VPN environment for highly secure application requirements.**

Client and Listener

Listener component built within the “client” is similar to the server component and enables application environments to use publish-subscribe mechanism. Listener enables peer to **peer and grid deployment** capabilities. Listener along with the scheduler can be used for **application wide configurable “alerts”**.

Object Processor Engine

The “Object Processor Engine” is an innovative application execution environment.

Availability of distributed object dictionary enables retention of object information until the execution time. It works on the philosophy of **Complex Instruction Set Computer**, wherein most of the functionality is available in the form of native code. Only in case of dynamic resolution of object references and expression evaluation, byte code is generated and executed.

Additionally, **fast assign and fast move** technologies are incorporated in the processor engine for maximum runtime efficiency, wherein assignments are handled directly, than using stack-based approach.

In Object oriented runtime environments, every thing is a class and a garbage collector is required to release to memory. In Vaakya, the framework based micro-components, object dictionary and the Object processor together ensure that memory for objects not in use are immediately released there by reducing memory requirements and providing better performance.

Advantages

- Better efficiency
- Dynamic object/component resolution
- No need for application wide garbage collection
- Reduced memory requirements

Synchronization Engine

In any distributed software environment, the key issue is to keep **runtime, application source code and data** in sync. The object aware synchronization engine is a portable technical component used for updating the runtime and application source in a distributed environment.

Data synchronization is handled by an innovative; transaction oriented, rule based data synchronization mechanism. (Please refer to the Business application framework for details) This addresses the need for distributed data storage requirements.

Advantages

- The object aware data synchronization mechanism consumes very less storage space
- **Data transfer is based on objects as a whole rather than the conventional record based approach**
- Runtime and application synchronization reduces the migration and change management issues

Workflow and Scheduler Engine

Workflow and Scheduler component is a necessary part of any distributed, dynamic software application.

The Vaakya Workflow Server is a generic component that allows flexible workflow configuration and provides value-based rights to objects and components.

Scheduler and listener in combination allow configuring synchronous or asynchronous alerts.

Built-in encryption and digest mechanism provides secure and authentic transaction management environment.

Advantages

- **Publish – Subscribe for events and alerts on runtime, source and data.**
- Message broadcasting
- **Task based business process automation**

Object/Micro-Component Locator

The Component locator identifies the location/type of remote objects and components at runtime. These components may reside on different systems and are configurable.

Vaakya components are accessed using message requests over the Vaakya Communication Protocol.

Request and response for non-Vaakya components are achieved through XML Web services interface.

Advantages

- Flexible deployment capability
- Runtime **dynamic object loading**
- Interoperability using XML Web services and Vaakya API's

Memory Manager- “Producer – Consumer” Model

Vaakya memory management model uses a balanced approach of optimal resource usage for both built-in micro-components and user-defined functions.

Micro-Components

Micro-components manage all memory allocation and freeing functions. These components are self-managing and programmers need not handle memory allocations.

User Defined Functions

User defined functions in Vaakya environment call for a different approach as explained below:

Conventional approach

Let's see an simple example of 2 functions.

```
function1()
{
  x = function2() //call to function2
}

int function2()
{
  x = new(y)
  return x;
}
```

- Freeing memory of variable y can be handled within function2.
- Freeing memory of variable x in function2() has to be handled by the programmer (C, C++) or by employing application wide garbage collection mechanism (Java).

Vaakya Producer-Consumer Model

In Vaakya scripting language, developers need to allocate memory but the producer-consumer model of memory management technique frees the memory automatically.

```
function1()
{
  x = new(); // memory allocation
  function2(x) ; // x is passed as a parameter
}

function2(x)
{
  x = y; // No return statement
}
```

In this model,

- Memory for x is allocated in function1 and parameter passed on to function2
- Function2 frees its stack on return
- Function1 will free the memory of x on completion of itself.

Notice that a) there is no return statement in function2 and b) x is passed as a parameter to function2 rather than using an assignment [x = function2()] in function1.

Advantages

- **There is no need for application wide garbage collection mechanism**
- Minimal memory requirement
- Developers need to allocate memory only for user defined functions, but need not free the memory.

Security Components

Distributed and dynamic applications call for higher level of authentication and security from different perspectives like application access, security for application **source code, data at rest (Stored in data store or other DBMS) or data in transit.**

Industry standard, digest and encryption mechanism is used for securing the source code and data. Vaakya Object Server offers column level data encryption mechanism for securing data.

The built-in security components provide a communication protocol independent secure communication capability.

External Interface

Message Oriented - Vaakya Communication Protocol

Remote method invocation is handled using a **socket based, message-oriented** Vaakya Communication Protocol (VCP).

Micro-components can be located and invoked using the communication interface. Messages are transferred using VML (Vaakya Markup Language).

VCP can be used for connecting applications in a dynamic, static, synchronous or asynchronous mode over LAN, WAN, HTTP, GPRS SMS, MMS or SMTP.

Messages are encrypted and secured with digital signature for security. Messages beyond a specific size are compressed and transferred across the network.

Additionally Vaakya API (Application programming interface) can be used for binary integration of applications.

Advantages

- Text based messages allow Vaakya applications to be deployed on heterogeneous environments
- Interoperability through XML web services
- **Dynamic switch, over the underlying communication protocol provides fault tolerance**

Component Integration

- **API**

Vaakya framework is completely built using “C” programming language. Most language environments provide interfaces to C components.

- **Binary standards**

COM, CORBA and other RPC implementations provide binary interface for component integration. Support for these standards in Vaakya environment will be market driven.

- **XML Web services**

Vaakya is built on XML based **message-oriented** remote method invocation and hence can be extended to support any of the SOAP protocol interfaces for XML Web services based component integration methods.

Data Integration

- Live updates through ODBC or native driver connectivity

ODBC and SQL (Structured Query Language) standards compliance is one of the basic data integration methods.

Vaakya environment is **model- driven** (Distributed Object dictionary) and can be extended through ODBC or native database drivers to support industry standards.

- Batch updates through XML or character delimited (Comma, Pipe etc) formats

Batch updates largely use character delimited and XML based data exchange. Vaakya provides both these capabilities for data exchange.

Application Development

Application development model on Vaakya environment for different domain frameworks is explained in greater detail in the Vaakya Business Applications framework whitepaper and Vaakya tutorial.

Application Deployment

All that we need to deploy a Vaakya application is, a VXE (Vaakya Executable), Vaakya runtime and a set of configuration files.

VXE

The source code for Vaakya applications in the form of text is compressed, encrypted, digitally signed and stored in the form of a VXE.

When an application is launched, the signature is verified based on the user/vendor and then it is decrypted, uncompressed and loaded.

Type checking is performed on loading the script and hence provides better performance as against conventional interpreted languages that resolve types at runtime.

Runtime

Vaakya runtime contains all the framework specific technical and functional components necessary for executing the application. **Vaakya runtime can serve and handle multiple applications at the same time.**

Configuration Files

Configuration files are provided for application, client and server; basically used for logical to physical mapping of the following

- Screen resolution
- Fonts and sizes
- User variables
- Location of components and objects
- Connection mode (Static, Dynamic)

Deployment Modes

- Standalone
- Client-server
- N-tier

Application Behavior

- Peer-2-peer, Grid
- Online, Offline, Hybrid (online and offline)

Connectivity

Vaakya applications can be connected to server in a dynamic, static, synchronous or asynchronous mode. Even on a LAN, users can connect to the server in a dynamic mode (similar to http protocol where connection is closed after each request/response). This mechanism allows for serving maximum number of users with minimal resource requirements.

Advantages

- Developers need not design and develop their application for various deployment models.
- Static, dynamic, synchronous and asynchronous connectivity over underlying communication protocols

Application Execution

Vaakya applications need Vaakya runtime for application execution. The runtime is an **Object Interpreter** (not a byte code interpreter) made possible by the combination of **framework based native code** implementation and the **distributed object dictionary**.

Standalone

- Vaakya script
- Runtime
- Application configuration
- External component libraries

Client-Server

- **Runtime should be available in all systems using Vaakya applications**
- Vaakya component locator
 - Application configuration
 - Server configuration
 - Client configuration
- Source code (Vaakya Script) is centrally located in the server
- External component libraries

Advantages

- Vaakya Applications consume very less storage space and are designed for minimal memory and network resource usage

Summary

Framework-Oriented application development using “**Vaakya Distributed Computing Architecture**” offers major benefits for distributed, dynamic and heterogeneous software applications as against generic, interpreted or monolithic methods.

- **Development**
 - Integrated Solution
 - Framework based Descriptive scripting
 - Model Driven Architecture
 - Selective micro-component feature assembly
- **Deployment**
 - Cross platform, Device neutral
 - Flexible deployment over LAN, WAN, HTTP, GPRS, SMS, SMTP
 - Online, offline or hybrid deployment
 - Standalone, client server or peer-to-peer configuration
 - XML Web services
- **Runtime**
 - “Object Aware” application execution
 - Message-oriented and secure communication
 - Efficient memory and bandwidth usage
 - Rich client user interface
 - Fault tolerance
 - Tiny foot print and reduced layering

The **small foot print** integrated approach along with the distributed object dictionary, allows deploying applications in a distributed environment for grid or peer-to-peer.

Applications can be configured to execute as online, offline or hybrid mode due to locally available data store and other micro-components.

Updates to the Vaakya runtime will eliminate the need for application migration and version management.

Collaborative application modeling environments built around Vaakya architecture enable programmers/domain specialists to describe and change their applications.

Integrated, small footprint, configurable Vaakya runtime enable even PDA's, Smart phones and USB keys (possibly even smart cards) to act as server computing environments or intelligent devices.

Terminology

Distributed

When computer programming and data on which computers work are spread out over more than one computer, usually over a network, then it is termed as "distributed". It may be categorized into computation intensive or distributed data environments.

Heterogeneous

A common use of this word in information technology is to describe a product as being able to contain or be part of a "heterogeneous network," consisting of different manufacturers' products that can "interoperate." The Internet itself is an example of a heterogeneous network.

Dynamic

Software applications evolve over a period of time. The dynamic nature would represent change in business process with corresponding change to underlying data and distribution of components from one location to another.

Software Architecture

Software architecture is the higher-level structure of a software system. The structure must support the functionality required by the system. Thus, the dynamic behavior of the system must be taken into account when designing the architecture.

The structure--or architecture--must conform to the system qualities (also known as non-functional requirements). These include performance, security and reliability requirements associated with current functionality, as well as flexibility or extensibility requirements associated with accommodating future functionality at a reasonable cost of change. Conflicts and tradeoffs within the requirements are an essential part of architecture design.

Framework

A framework is often a layered structure indicating what kind of programs can or should be built and how they would interrelate. Some computer system frameworks also include actual programs, specify programming interfaces, or offer programming tools for using the frameworks. A framework may be for a set of functions within a system and how they interrelate; the layers of an operating system; the layers of an application subsystem; how communication should be standardized at some level of a network; and so forth.

Model Driven Architecture

MDA development focuses first on the functionality and behavior of a distributed application or system, undistorted by idiosyncrasies of the technology platform or platforms on which it will be implemented. In this way, MDA divorces implementation details from business functions. Thus, it is not necessary to repeat the process of defining an application or system's functionality and behavior each time a new technology (Web Services, for example) comes along. Other architectures are generally tied to a particular technology. With MDA, functionality and behavior are modeled once and only once.

Portable

Applications should be deployable across different operating systems, processor types and devices without need for recompilation.

Patterns

In software development, a pattern (or design pattern) is a written document that describes a general solution to a design problem that recurs repeatedly in many projects. Software designers adapt the pattern solution to their specific project. Patterns use a formal

approach to describing a design problem, its proposed solution, and any other factors that might affect the problem or the solution.

Components

A component is an identifiable part of a larger program or construction. Usually, a component provides a particular function or group of related functions. In programming design, a system is divided into components that in turn are made up of modules.

Integrated

“Integrated” represents a combination of tools and utilities that are provided to solve a specific problem. Utilities like source editor, compiler, debugger and version control system represent an integrated development environment. Similarly Vaakya environment provides utilities like data store, application server, IDE etc.

Message Oriented

Message-oriented middleware is a client/server infrastructure that increases the interoperability, portability, and flexibility of an application by allowing the application to be distributed over multiple heterogeneous platforms. It reduces the complexity of developing applications that span multiple operating systems and network protocols by insulating the application developer from the details of the various operating system and network interfaces- Application Programming Interfaces (APIs) that extend across diverse platforms and networks, typically provided by the MOM.

Descriptive Scripting

Developing software applications using generic programming languages are complex and time consuming. This involves memory management; thread handling, use of appropriate data structures and algorithm development. Descriptive scripting follows a framework-based approach that hides most of the technical complexities involved in developing applications.

Micro-Components

Independent components that are be dynamically loaded or extended to runtime without affecting other components.

Producer – Consumer

Producer – Consumer model is a concept of memory management in Vaakya. The producer is responsible for allocating and freeing the memory for the objects used. The consumer just uses the memory and cannot free the memory allocated by the producer.

Object

An object represents a real world entity identified by its properties and behavior. But in software terms, Objects are contextually used to represent a function or a component.

Object Processor Engine

Object processor engine in Vaakya runtime environment executes the Vaakya script. It is a combination of native code execution and byte code interpretation in an object aware environment. It differs from the conventional hybrid environments due to the presence of predefined component structures and distributed object dictionary.

Object Dictionary

Object dictionary is a collection of data object definitions and their relationships that reside in memory. This is distributed across all the systems that execute Vaakya application. The contents of object dictionary would depend on the application domains like GIS (Geographical Information Systems), GPS (Global Positioning Systems), Graphics/Image processing etc.

Abbreviations

Term	Description
API	Application Programming Interface
DBMS	Data Base Management System
DDL	Data Definition Language
DML	Data Manipulation Language
IDE	Integrated Development Environment
MDA	Model Driven Architecture
MOM	Message-Oriented Middleware
VQL	Vaakya Query Language
RAM	Random Access Memory
VAM	Vaakya Application Manager
VCP	Vaakya Communication Protocol
VGE	Vaakya Graphics Engine
VAS	Vaakya Application Server
XML	Extended Markup Language
VML	Vaakya Markup Language
VXE	Vaakya Executable

Appendix

VML Construct

```
<VML>
  <VML_RECORD = Employee, 0>
    001~ (empname) ~ (dept) ~ (designation)
  </VML_RECORD>
</VML>
```

Note : Structure of the employee object as specified in the tag "<VML_RECORD>" is available across all systems using Vaakya application. Hence, the property tags and data types need not be created and parsed every time.

Contact Us

Corporate Office:

601, First Floor,
12th Main, HAL 2nd Stage,
Indiranagar, Bangalore
PIN - 560008
India

feedback@vaakya.com

Copyright

"The copyright of any and all material contained herein is owned and reserved by VAAKYA TECHNOLOGIES PRIVATE LTD ("**OWNER**") except where otherwise stated. Any reproduction, copying and/or distribution in any form of the material, in whole or in part, is not permitted without prior written consent from the OWNER. Trademarks, logos, images, text or content of third parties ("**THIRD PARTY PROPERTY**") used herein are the property of their respective owners, and have been used without permission except where otherwise indicated. The Owner makes no claim to such THIRD PARTY PROPERTY used herein. All use of THIRD PARTY PROPERTY contained herein is for non-profit purposes only, for the proper guidance of the users, considering the convenience and familiarity of the user in associating the THIRD PARTY PROPERTY with the owners of THIRD PARTY PROPERTY and not for advertisement or establishing any connection with the owners of THIRD PARTY PROPERTY. Other than THIRD PARTY PROPERTY, all trademarks, tradenames, logos, designs and all related product and service names are the sole property of the OWNER, and may not be used in any manner without the prior written consent of the OWNER. The materials contained herein may include inaccuracies or typographical errors and the owner shall not be held responsible for the same. The Owner reserves the sole right to modify, amend, delete, omit, edit or include any material or make periodic changes to the material herein.