USC CSCI 578 – Software Architectures
Fall 2025
Homework Assignment #1
Due: *Thursday, September 25, 2025*
– see Brightspace for submission details –

For this assignment, you must choose one of the two case study assignments provided below: **C4** or **BOINC**. There are two Appendices I and II, each of which describes the corresponding case study pertaining to a large-scale software system.

Your assignment will be slightly different depending on your choice of case study, but both options are comparable in difficulty. These case studies aim to help you understand how to do the following:

- Decompose the description of a desired system and its requirements into components and connectors.
- Understand how the designs you come up with typically exhibit one or more architectural styles.
- Weigh the tradeoffs of selecting one architectural style over another.
- Understand how your design succeeds or fails in satisfying requirements and addressing architectural challenges.
- Explain how a use-case scenario relates to the architecture that you design.

Your choice of a particular case study will in no way impact your grade on this assignment. Solving more than one assignment is not expected and will not result in any extra credit. Please focus on a single case study.

You must clearly indicate which case study you have selected.

Note that there is no limit on which "language" to use when drawing the diagrams. You are recommended to use diagram drawing software. You may also draw with a pen on paper and then scan it. In any case, the diagrams must be clear and readable to the grader.

This assignment involves "architectural styles." You should have a solid understanding of architectural styles after this week's lecture, which focused specifically on them. For your reference, Appendix III describes the styles that the course textbook introduces.

C4 Case Study Assignment

The Call Center Customer Care (C4) Case Study provided as an appendix to this assignment presents an initial high-level ("Level 1") architectural breakdown for the system used by a large telecommunications company. This system comprises several subsystems, one of which is C4 itself.

1. The lectures and readings have begun discussing the architectural design process, including the selection of architectural styles. Not all styles are applicable to all systems, and any choice of style will involve making trades on various system attributes. Pick any two styles that you have read about and design two different architectures for the C4 system, one that adheres to each style. In order to apply a style, you will need to create a detailed architectural breakdown (a "Level 2" architectural breakdown) for C4. In other words, "expand" the C4 box shown in the figure on page 1 in the Case Study into an architecture. It is, of course, difficult to decide on the exact degree of detail to be provided in a "Level 2" architecture, such as the one required for this assignment. Make sure to show all the connectors inside the C4 architecture as well as those that interconnect C4 to the other parts of the system. Moreover, you should graphically distinguish different types of connectors used in your design. Also, there is no such thing as the "correct" or "optimal" architecture. However, as a granularity guideline, your decomposition of C4 should consist of no less than 20 and no more than 30 distinct components.[1] You are not required to select a style from the course textbook, but you must let us know what style you are attempting to apply and provide a reference if the style is not one from the course text.[2] **Submit one diagram for each architecture you design.**

2. Give a brief rationale for your architecture: why did you select the styles that you did? Suggest two pros and cons of each architectural style and weigh them. We will not grade you based on how accurately you apply each style so much as your rationale for selecting a particular style and understanding its limitations. **Please limit your answer to 2 paragraphs for each style (*i.e.*, a total of 4 paragraphs).**

3. Compare each of your architectures: give an example or two of a system property or requirement described in the Case Study that one of your architectures addresses in a superior manner. Be sure to not only name the property or requirement but also explain how each architecture addresses the property or requirement and give the rationale for why you think one system is superior to the other. **Please limit your answer to 2 paragraphs.**

4. Since C4 is a vast system with many different, possibly conflicting, requirements, your architecture may only directly address a subset. To demonstrate this, for one of your

[1] This is only a guideline. There is no magic about this number, nor do we have a specific preferred solution in mind.

[2] When you provide references, use the IEEE reference style: https://journals.ieeeauthorcenter.ieee.org/wp-content/uploads/sites/7/IEEE_Reference_Guide.pdf.

architectures, select three of the key architectural challenges and requirements (listed in the C4 Case Study appendix) and argue and discuss how your architecture DOES NOT support them in an acceptable fashion. **Please limit your answer to 2 paragraphs.**

5. Modify one of your architectures to address the shortcomings discussed in the preceding question. Can you do so without violating the chosen architectural style? Give the rationale for whether you think this change is appropriate for the system. **Please limit your answer to 1 paragraph and a new diagram.**

6. Add a capability to both your architectures to log interactions between C4 and Downstream Systems. Logging should be enabled only when there are less than 100 active users interacting with the system concurrently. How will you change each architecture in response to this new requirement? Which architecture can be changed more easily? Why? **Please limit your answer to 2 paragraphs and, if you find it necessary, one diagram for each of your architectures.**

BOINC Case Study Assignment

The Berkeley Open Infrastructure for Network Computing (BOINC) Case Study, provided as an appendix to this assignment, presents an initial high-level ("Level 1") architectural breakdown for the system. The overall BOINC system comprises several components, one of which is the BOINC task server.

1. The lectures and readings have begun discussing the architectural design process, including the selection of architectural styles. Not all styles are applicable to all systems and any choice of style will involve making trades on various system attributes. Pick any two styles that you have read about and design two different architectures for the BOINC task server, one that adheres to each style. In order to apply a style, you will need to create a detailed architectural breakdown (a "Level 2" architectural breakdown) for the task server. In other words, "expand" the task server box shown in Figure 4 in the Case Study into an architecture. Make sure to show all the connectors inside the task server architecture as well as those that interconnect the task server to the other parts of the system. Moreover, you should graphically distinguish different types of connectors used in your design. Also, there is no such thing as the "correct" or "optimal" architecture. However, as a granularity guideline, your decomposition of the task server should consist of no less than 20 and no more than 30 distinct components. [3] You are not required to select a style from the course textbook, but you must let us know what style you are attempting to apply and provide a reference if the style is not one from the course text. **Submit one diagram for each architecture you design.**

2. Give a brief rationale for your architecture: why did you select the styles that you did? Give two pros and cons, and weigh them for each architectural style. We will not grade you based on how accurately you apply each style as much as your rationale for selecting a particular style and understanding its limitations. **Please limit your answer to 2 paragraphs.**

3. Compare each of your architectures: give an example or two of a system property or requirement described in the Case Study that one of your architectures addresses in a superior manner. Be sure to not only name the property or requirement but also explain how each architecture addresses the property or requirement and give the rationale for why you think one system is superior to the other. **Please limit your answer to 2 paragraphs.**

4. Since BOINC is a vast system with many different, possibly conflicting, requirements, your architecture may only directly address a subset of those requirements. To demonstrate this, for one of your architectures, select three of the key architectural challenges and requirements (listed in bulleted items in the last two pages of the Case Study) and argue/discuss how your architecture DOES NOT support them in an acceptable fashion. **Please limit your answer to 2 paragraph.**

[3] This is only a guideline. There is no magic about this number, nor do we have a specific preferred solution in mind.

5. Add a capability to both your architectures to integrate one or more cloud(s) into the BOINC computation. The project operator must be able to vary the amount of cloud computation power involved in the BOINC computation based on the active volunteer-participant numbers. How will you change each architecture in response to this new requirement? Which architecture can be changed more easily? Why? **Please limit your answer to 2 paragraphs and, if you find it necessary, one diagram for each of your architectures.**

Appendix I

The Call Center Customer Care System (C4)

(a case study)

***Note*: this is a simplified and generalized description of a real system**

Introduction

The *Call Center Customer Care System* (C4) has been developed by Andersen Consulting for a large US telecommunication company. The primary function of the system is to support interactions with the customers that request new services (ex: new phone lines), changes in the configuration of the existing services (ex: phone number changes, long-distance company changes, or relocation), or report problems.

The phone company has over 19 million customers. Considering how often, on average, a customer changes his/her service configuration, the system has to support up to 400 company representatives simultaneously at near 24/7 availability level. However, these representatives are not the only means by which a customer can request a change or report problems. For example, there exists a phone service (Quick Service) by which customers can communicate with the system. This is discussed in more detail later.

System Interactions

The C4 system interacts with a number of other systems, in particular with:

- A network provisioning system that makes physical changes to the network configurations and supports network management (depicted as "NOSS" in Figure 1 below)
- A Billing system
- A host of corporate databases (DBs), and
- A number of downstream systems.

A high level structure of the system interactions is show in Figure 1. Collectively, this is a "big thing" and rather complicated.

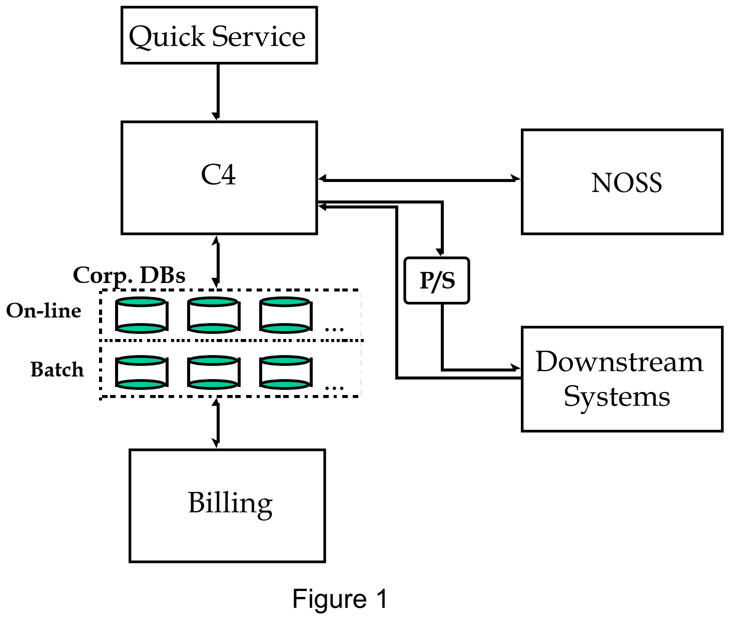

Figure 1

Interactions with NOSS

NOSS (Network Operations Support System) is the network management and provisioning system. It is being developed concurrently by a large 3rd party hardware/software company specializing in communication networks. Its functionality contains:

- Work force management - management of maintenance crews
- Provisioning - putting physical network components in place such as connections from the curb to the house
- Network creation - an peculiar name for maintaining information about new and existing physical network
- Activation - automatic turning on/off of services
- Network management that contains:
    - status monitoring and measurements
    - proactive maintenance
    - diagnostics, and
    - problem reporting, and
- Field access - a subsystem providing up-to-the-minute information for field technicians.

The NOSS architecture is based on the OSI CMIS (Common Management Information Service). The interface between C4 and NOSS is a large set of messages in the named-tag/value format. Each message inter-change is technically a synchronous pair of messages composed of a request and a reply. At the application level, there are two types of interactions: 1) a request followed by a response that contains the requested information and/or confirmation of NOSS action, and 2) an overall interaction consisting of two message pairs; a) initial request followed by a reply containing a request identifier, b) an unsolicited message from NOSS containing the previously issued request identifier along with related data followed by a C4 reply acknowledging receipt of message.

The basic interactions between C4 and NOSS are as follows:

- C4 sends a Service Request (service order messages) to NOSS to perform service reconfigurations
- C4 send queries to NOSS about existing network status or capabilities. This is usually done while an agent is talking to a customer. For example, a request may be sent to NOSS about availability of service at a particular geographic address/service location or possible service activation dates, and
- C4 may request from NOSS to "lock" resources such as phone numbers for a service request.

Interactions with Downstream Systems

Downstream Systems are systems such as: Long-distance Carrier Services, 911 Service, Voice Mail, Interfaces to Phone Directory, and Revenue Collections (credit scoring and checking).

The C4 interacts with these systems in two ways:

- It publishes requests via the Publish/Subscribe (P/S in Figure 1) component that the downstream systems should react to. For example, all new phone connections are published so the 911 service is connected to them in the required number of hours
- The downstream systems notify C4, via a direct asynchronous message, about significant business event that effect customer configuration. For example, the Long-distance Provider system may notify C4 that a client should be connected to a new long-distance company.

Note: one of the design challenges that we will discuss later was that direct requests from a customer may conflict with a similar request coming from an external system. For example, a customer may call to request a change of their long-distance carrier, say from Sprint to MCI, at the same time ATT sends notification that the same customer has now selected them as their long-distance carrier.

Interactions with corporate DBs and Billing

Some of the major functions of the billing system are: (1) invoice calculation for local services, (2) invoice printing for both local and long-distance services, (3) revenue reporting, and (4) bill inquiry and adjustment.

C4 does not interact with the Billing System, but it works against the same set of DBs. More specifically, C4 interacts with the on-line part of the corporate DBs, while the Billing Systems work with the batch copy of the on-line DBs (this is shown in Figure 1). Monthly billing is done in a number of cycles. Each cycle processes billing information of a set of customers. All updates to the data of the current cycle are halted (applied only to the on-line data) until the end of the cycle. The updates (at the end of the cycle) are done in batch and are transparent C4.

All customer data are stored in over 100 tables. C4 does not use all of them. During any customer conversation, C4 obtains customer basic profile from about dozen tables. Other tables are read and/or updated on demand.

C4

Functional description

C4 is an online transaction processing (OLTP) system that handles an interesting type of transactions. A transaction is initiated by a customer call, or so called Business Event. There are three types of events:

- Service Negotiations
- Account Management, and
- Trouble Call Management.

Each event is then divided into Tasks which are in turn broken into Activities. Tasks are groups of related activities that the representative and the customer have to complete. For example, if the negotiation is about the customer moving from one address to another, there will be a set of activities related to terminating some of the existing services, a set of activities related to obtaining the new address, and a set of activities related to negotiating new services and activation dates.

C4 must provide:

- Support for multiple concurrent tasks. For example, the customer should be able to negotiate a number of different services during the same call.
- Integrated support for completing activities (screen sequences, to-do list, context-sensitive data fields, etc.)
- Validation
    - of availability of requested service
    - completion of activities and tasks
    - integrity of customer data, and
    - integrity of the final requested configuration.
- Advice on available products and product "bundles"
- Resolution of conflicting events, and
- Support for interrupted and long-lasting conversations.

The last two points are particularly interesting. Resolution of conflicting event comes from multiple so-called Authors of events. For example, while a wife is negotiating a new phone line, the husband is using the phone company kiosk at a bank to request an ISDN line that will have an extra two phone lines. In general, events can come from:

- Direct conversations with company representatives (the case described here)
- Automated call center (the phone menu-type system)
- Kiosks (future), and
- Direct customer connections such as Internet (future).

Before C4 sends a service request to NOSS, it has to make sure that all related events have been combined or conflicts resolved.

The other requirement comes from the fact that a conversation with a customer can be interrupted (for technical reasons, for example) or suspended by the customer or the representative. The first case is rather obvious. An example of the second case when a customer that says something like “let me talk to my wife and I will call you back”. In any case, C4 has to manage the context that persists and can be recalled.

Key architectural challenges

Here is a partial list of architectural challenges for C4. The challenges are not completely independent, so there is some repetition in the list. A number of challenges are implied by the execution architecture selected for the system. More about the architecture in the following section.

- Managing time and date effectively. Customer may want service changes in the future and this implies:
    - some form of a tickler system
    - ability to inform the customer about future changes of rates and/or services
- Interfacing with multiple authors of business events. As explained above, the main source of business events is direct conversation of an agent with a customer. However, other sources such as downstream systems, kiosks, etc. have to be accommodated. Important points:
    - business events from different authors may conflict with regards to the requested configuration at a service location
    - business events from different authors may be received and processed at the same point in time – business events from different authors become different service requests that must be cross validated to ensure a valid resulting configuration
- Architect something that is economically viable with a small set of customers and yet can grow to a very large network (i.e. 15 million customers)
    - it should not require high initial equipment investment
    - it should allow for “leaner” growth (in respect to cost(capacity) function)
    - it should be able to grow at a rapid rate (for example, 1000+ new customers a day)
    - identification, monitoring, and elimination of processing bottle-necks
- Validation of a requested service configuration should be done at near-real-time. This implies that:
    - C4 has to communicated with NOSS and other systems while guiding agents through tasks and activities, and
    - C4 may request to “lock” some of the network resources for a fixed amount of time (like a few phone numbers)
- Support for long-lasting, interrupted sessions. This issues has been described above
- Integrated (smart) performance support for company representatives
- Support a large number (e.g. 400+) of service representatives concurrently
    - a minimum of 100 service representative to start
- Near 24/7 application availability

Execution architecture

The system execution architecture is a standard three-tier client/server configuration as shown in Figure 2.

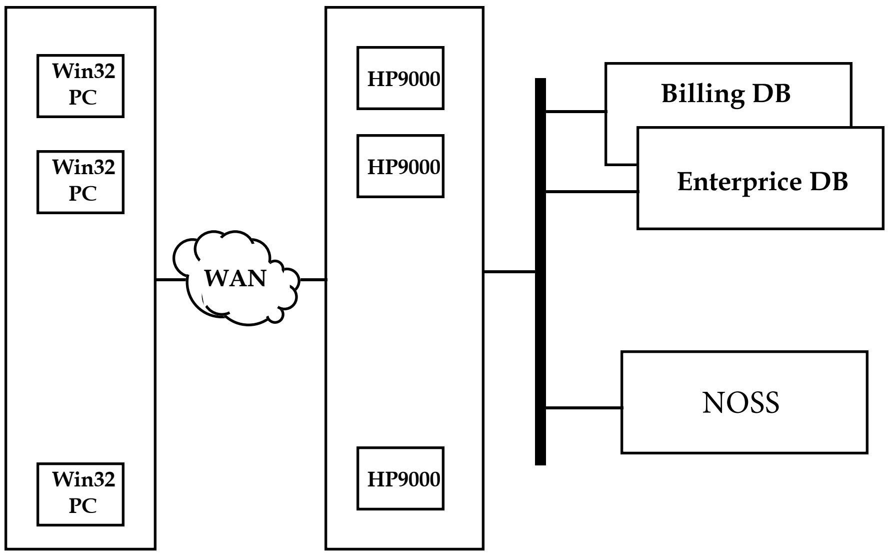

Figure 2

All agents workstations are PCs running the WinNT OS connected to a LAN that is interconnect to a WAN. The middle layer is a cluster of HP9000 servers running UNIX and a TP Monitor that can load-balance between them. The TCP/IP communications protocol is used throughout the network. The back-end runs on a dedicated high throughput LAN. This LAN connects the Enterprise and Billing DBMSs to the servers as well as a TCP/IP connection to NOSS via a module that marshals messages between C4 and NOSS. C4 runs on the cluster of PC and HP9000.

Additional architectural requirements are as follows:

- No persistent data caching on the agent workstations to limit the implications of local failures
- No DBs at office locations
    - no administrators at local offices
    - no maintenance down-time, etc.
- The middle layer server cluster tuned for performance
    - possibility to add serves to increase throughput
- The back end tuned for DB performance
    - preferred place to do persistency
- Well engineered operations architecture
- High availability cannot be achieved by utilizing fault-tolerant hardware (this option is not economically viable)

Appendix II

Berkeley Open Infrastructure for Network Computing

(A case study)

Note: this is a simplified and generalized description of a real system

Introduction

The *Berkeley Open Infrastructure for Network Computing* (BOINC) is an open-source software for volunteer computing and grid computing. The primary function of the system is to make it possible for researchers to tap into the enormous processing power of personal computers around the world.

BOINC was originally developed to support the SETI@home project (https://setiathome.ssl.berkeley.edu), an experiment that analyzes radio telescope data from outside the Earth as an attempt to find extraterrestrial intelligence. The challenge was that the analysis of radio telescope signals required enormous amount of computation resource as the received signals consisted of much noise and man-made signals. The more computation resource involved, the wider frequency range coverage with more sensitivity[1]. Instead of adopting high-cost, supercomputers for the computation, a high performance distributed computing platform, BOINC, was developed to take advantage of the idling cycles of personal computers of people around the world who wanted to participate in the project.

Overtime, BOINC has become useful as a platform not just for searching for extraterrestrial intelligence but also for other distributed applications in areas as diverse as mathematics, medicine, molecular biology, climatology, environmental science, and astrophysics. It has about 320,121 active participants and 512,197 active computers (hosts) worldwide processing on average 6.6 petaFLOPS as of July 23, 2014.

The distributed nature and the gigantic scale of BOINC bring up several design challenges. For example, it must be able to manage a large pool of participating nodes (personal computers) remotely via computer networks. There could also be performance, security, and/or scalability concerns when designing its architecture. This is discussed in more detail later.

Volunteer Computing: How It Works

Participant's Perspective

Volunteer computing performs computation using participants' computation resource. Figure 1 depicts the participant's perspective of a volunteer computing system. The following steps are indefinitely iterated:

(1) A participating node (denoted as "Your PC" in the figure) receives a set of instructions,
(2) The node downloads the executable applications and the input files for computation,
(3) The node performs the computation and produces output files,
(4) The node uploads the output files to the server, and
(5) The node reports the list of completed tasks to receive reward and to adjust the size of future jobs.

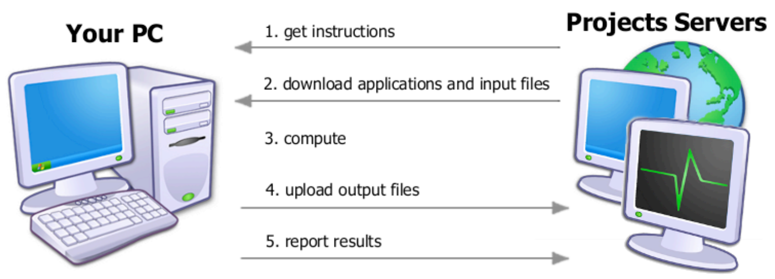

Figure 1: General Depiction of Volunteer Computing from the Participant's Perspective

[1] Refer to https://setiathome.berkeley.edu/sah_about.php for more information.

Project's Perspective

Volunteer computing provides an opportunity to projects that need large computation power and/or storage via collecting the resources from volunteers who wish to participate in the project. While the specifics of how a computation is performed differs system to system, but the high-level computational model (i.e., how the computation is divided, performed, and assimilated) is identical. Figure 2 depicts a volunteer computing system model in project's perspective. The task server receives a *computation* as its input. The computation is divided into multiple computation *tasks*. A task is a part of the computation that is independent from the other tasks. Each task is duplicated into redundant *jobs*, which are the ones transferred in the second step of Figure 1. With redundancy, each task is executed as several identical jobs on distinct nodes. The created jobs are assigned to a randomly selected participant node via the job queue. When the nodes report back the results of the assigned jobs (step 4 of Figure 1), the task server compares and accepts the results. New jobs could be created based on the results. This process is iterated until no more jobs are left in the job queue.

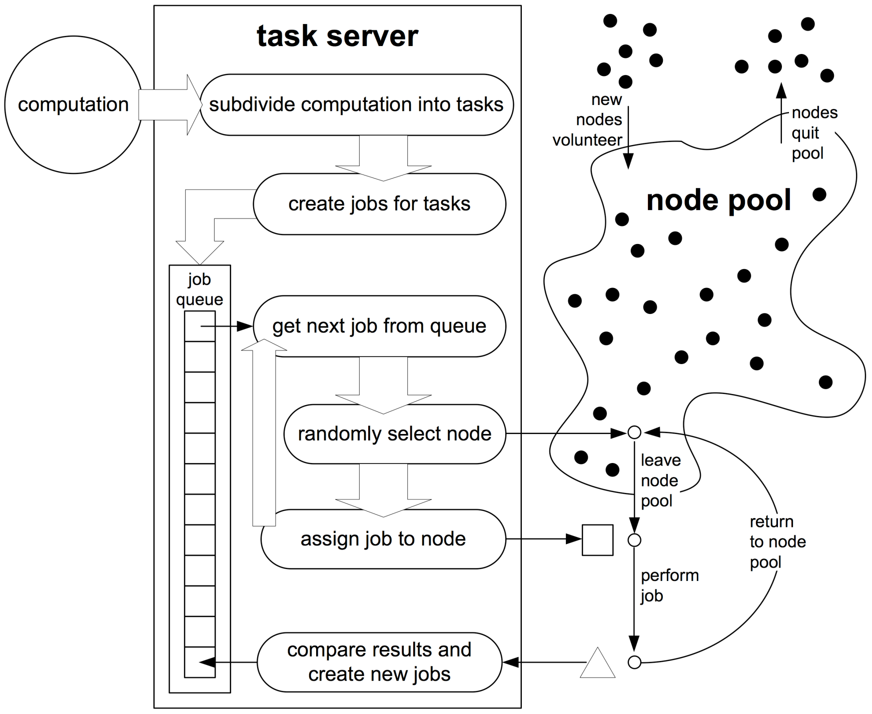

Figure 2: General Depiction of Volunteer Computing from the Project's Perspective

System Description

Overall System Description

BOINC has the general goal to nurture the public resource computing. It has the following specific goals:

- Reducing the barriers of entry to public-resource computing
- Sharing resources among autonomous projects
- Supporting diverse applications
- Rewarding participants

The Level 1 architecture (depicted in Figure 1) of BOINC is in the Client-Server architectural style. The architectural breakdowns of the BOINC client and server are in the following subsections.

BOINC Client

BOINC client consists of four major components and communicates with the components in BOINC server. Figure 3 depicts the BOINC client. The list of components is as following:

- The *schedulers and data servers* are installed on computers owned and managed by the projects to which the volunteers donate time of their computers.
- The *core client* communicates with the BOINC servers via the HTTP communications protocol to get and report work. The core client also runs and controls applications.
- *Applications* are the programs that do scientific computing. Several of them may run at the same time on a computer with more than one CPU.
- The *GUI* provides a graphical interface that lets the volunteers control the core client – for example, by telling it to suspend and resume applications. The GUI communicates with the core client via a TCP connection. Normally this is a local connection; however, it's possible to control a core client remotely.
- The *screensaver* runs when the participants are away from the computer. It communicates with the core client via local TCP, instructing it to tell one of the applications to generate screensaver graphics.

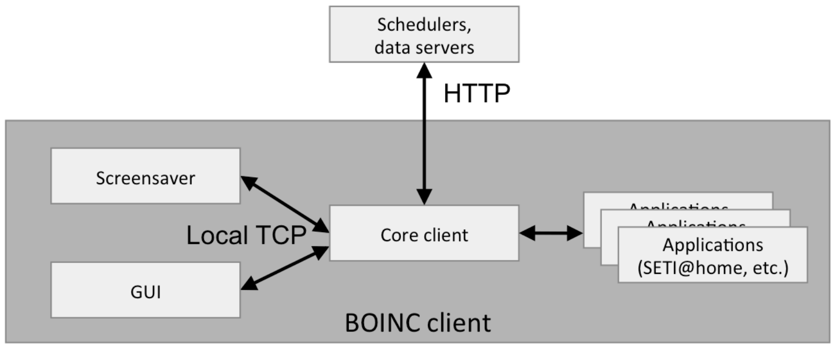

Figure 3: High-Level Architecture of BOINC Client

BOINC Server

BOINC server, including the task server, is depicted in Figure 4. It has three major components:

- *Web interfaces* for account and team management, message boards, and other features.
- *Task server* that creates tasks, dispatches them to clients, and processes returned tasks.
- *Data server* that downloads input files and executables, and that uploads output files.

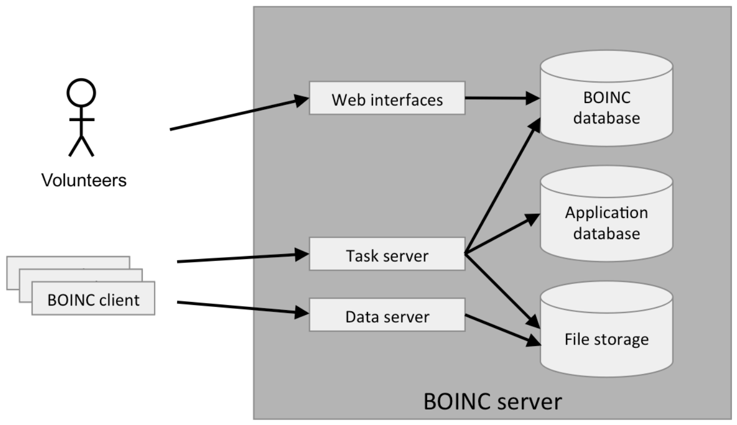

Figure 4: High-Level Architecture of BOINC server

BOINC Task Server Breakdown

The BOINC task server must have *at least* the following eight major components:

- The *work generator* creates new jobs and their input files. For example, the SETI@home work generator reads digital tapes containing data from a radio telescope, divides this data into files, and creates jobs in the BOINC database. The work generator sleeps if the number of unsent jobs exceeds a threshold, limiting the amount of disk storage needed for input files.
- The *scheduler* handles requests from BOINC clients. Each request includes a description of the host, a list of completed jobs, and a request for additional work, expressed in terms of the time the work should take to complete. The reply includes a list of jobs and their corresponding tasks. Handling a request involves a number of database operations: reading and updating records for the user account and team, the host, and the various jobs and instances.
- The *feeder* streamlines the scheduler's database access. It maintains a shared-memory segment containing (1) database tables such as applications, platforms, and application versions, and (2) a cache of unsent jobs. The scheduler finds jobs that can be sent to a particular client by scanning this memory segment.
- The *transitioner* examines tasks for which a state change has occurred (e.g., a completed job has been reported). Depending on the situation, it may generate new jobs, flag the task as having a permanent error, or trigger validation or assimilation of the task.
- The *validator* compares the instances of a job and selects a canonical instance representing the correct output. It determines the credit granted to users and hosts that return the correct output, and updates those database records.
- The *assimilator* handles job that are "completed": *i.e.*, that have a canonical instance or for which a permanent error has occurred. Handling a successfully completed job might involve writing outputs to an application database or archiving the output files.
- The *file deleter* deletes input and output files that are no longer needed.
- The *database purger* removes tasks and job database entries that are no longer needed, first writing them to XML log files. This bounds the size of these tables, so that they act as a working set rather than an archive. This allows database management operations (such as backups and schema changes) to be done quickly.

Key Architectural Challenges

Here is a partial list of architectural challenges of BOINC, the ones that must be considered for designing the BOINC task server. Note that the challenges are not completely independent; hence there could be some repetition in the list.

- BOINC must be resilient to erroneous computation results from nodes.
  - There could be malfunctioning computers or malicious nodes.
  - *Redundant computing* could be implemented to overcome this challenge. Replicate tasks into multiple jobs, distributed the jobs to random nodes, compare and accept a *canonical result* when the computation results are received from the nodes[1].
- BOINC server must be able to handle a large number BOINC clients.
  - It must be able to handle a large number of network connections with BOINC clients.

[1] This is called the Traditional Redundancy. Refer to [4] for more information.

    - The BOINC task server has to be able to process a large number of simultaneous jobs performed by BOINC clients.
- BOINC server must be able to credit rewards to participants for successful computation results (e.g., the results that have been accepted as the canonical result).
    - The reward could be in the form of points and ranking between the participants.
    - The rewarding system must be able to vary the amount of credit based on the amount of resource (computation, storage, and network transfer) the participants provided.
    - The rewarding system must be able to work across different projects and applications.
- BOINC server must be able to adjust the size of jobs it assigns to BOINC clients based on the clients' previous performance on the jobs that had been assigned to those clients.
- BOINC server must be available nearly 24/7.

References

This appendix is a summary of the following list of articles:

[1] Wikipedia: https://en.wikipedia.org/wiki/Berkeley_Open_Infrastructure_for_Network_Computing

[2] BOINC official website: https://boinc.berkeley.edu

[3] BOINC wiki page: https://boinc.berkeley.edu/wiki/How_BOINC_works

[4] Y. Brun, G. Edwards, J. Bang, and N. Medvidovic, Smart Redundancy for Distributed Computation, in Proceedings of ICDCS 2011.

[5] D. P. Anderson, E. Korpela, and R. Walton, High-Performance Task Distribution for Volunteer Computing, in Proceedings of e-Science 2005.

[6] D. P. Anderson, BOINC: A System for Public-Resource Computing and Storage, in Proceedings of GRID 2004.

Appendix III

USC CSCI 578 – Software Architectures
Fall 2025
Supplemental Lecture Material
– Software Architectural Styles –

This document introduces a list of software architectural styles and the relevant information about the styles we discuss in CSCI 578. It will come in handy when you work on the assignments and exams.

The architectural styles introduced in this document are primarily from the textbook, and the list is not exhaustive. There may be other architectural styles that we did not cover in this course.

Style: Main Program and Subroutines (no shared memory)

Summary: Decomposition based upon separation of functional processing steps.

Components: Main program and subroutines.

Connectors: Function/procedure calls.

Data elements: Values passed in/out of subroutines.

Topology: Static organization of components is hierarchical; full structure is a directed graph.

Additional constraints imposed: None.

Qualities yielded: Modularity: Subroutines may be replaced with different implementations as long as interface semantics are unaffected.

Typical uses: Small programs; pedagogical purposes.

Cautions: Typically fails to scale to large applications; inadequate attention to data structures. Unpredictable effort required to accommodate new requirements.

Relations to programming languages or environments: Traditional imperative programming languages, such as BASIC, Pascal, or C.

Style: Object-Oriented

Summary: State strongly encapsulated with functions that operate on that state as objects. Objects must be instantiated before the objects' methods can be called.

Components: Objects (aka. instance of a class).

Connector: Method invocation (procedure calls to manipulate state).

Data elements: Arguments to methods.

Topology: Can vary arbitrarily; components may share data and interface functions through inheritance hierarchies.

Additional constraints imposed: Commonly: shared memory (to support use of pointers), single-threaded.

Qualities yielded: Integrity of data operations: data manipulated only by appropriate functions. Abstraction: implementation details hidden.

Typical uses: Applications where the designer wants a close correlation between entities in the physical world and entities in the program; pedagogy; applications involving complex, dynamic data structures.

Cautions: Use in distributed applications requires extensive middleware to provide access to remote objects. Relatively inefficient for high-performance applications with large, regular numeric data structures, such as in scientific computing. Lack of additional structuring principles can result in highly complex applications.

Relations to programming languages or environments: Java, C++.

Style: Virtual Machines

Summary: Consists of an ordered sequence of layers; each layer, or virtual machine, offers a set of services that may be accessed by programs (subcomponents) residing within the layer above it.

Components: Layers offering a set of services to other layers, typically comprising several programs (subcomponents).

Connectors: Typically procedure calls.

Data elements: Parameters passed between layers.

Topology: Linear, for strict virtual machines; a directed acyclic graph in looser interpretations.

Additional constraints imposed: None.

Qualities yielded: Clear dependence structure; software at upper levels immune to changes of implementation within lower levels as long as the service specifications are invariant. Software at lower levels fully independent of upper levels.

Typical uses: Operating system design; network protocol stacks.

Cautions: Strict virtual machines with many levels can be relatively inefficient.

Style: Client-server

Summary: Clients send service requests to the server, which performs the required functions and replies as needed with the requested information. Communication is initiated by the clients.

Components: Clients and server.

Connectors: Remote procedure call, network protocols.

Data elements: Parameters and return values as sent by the connectors.

Topology: Two-level, with multiple clients making requests to the server.

Additional constraints imposed: Client-to-client communication prohibited.

Qualities yielded: Centralization of computation and data at the server, with the information made available to remote clients. A single powerful server can service many clients.

Typical uses: Applications where centralization of data is required, or where processing and data storage benefit from a high-capacity machine, and where clients primarily perform simple user interface tasks, such as many business applications.

Cautions: When the network bandwidth is limited and there are a large number of client requests.

Style: Batch-Sequential

Summary: Separate programs are executed in order; data is passed as an aggregate from one program to the next.
Components: Independent programs.

Connectors: The human hand carrying tapes between the programs, aka “sneaker-net."

Data elements: Explicit, aggregate elements passed from one component to the next upon completion of the producing program's execution.

Topology: Linear.

Additional constraints imposed: One program runs at a time, to completion.

Qualities yielded: Severable execution; simplicity.

Typical uses: Transaction processing in financial systems.

Cautions: When interaction between the components is required; when concurrency between components is possible or required.

Relations to programming languages or environments: None.

Style: Uniform Pipe-and-Filter

Summary: Separate programs are executed, potentially concurrently; data is passed as a stream from one program to the next.

Components: Independent programs, known as filters.

Connectors: Explicit routers of data streams; service provided by operating system.

Data elements: Not explicit; must be (linear) data streams. In the typical Unix/Linux/DOS implementation the streams must be text.

Topology: Pipeline, though T fittings are possible.

Qualities yielded: Filters are mutually independent. Simple structure of incoming and outgoing data streams facilitates novel combinations of filters for new, composed applications.

Typical uses: Ubiquitous in operating system application programming.

Cautions: When complex data structures must be exchanged between filters; when interactivity between the programs is required.

Relations to programming languages or environments: Prevalent in Unix shells.

Style: Blackboard

Summary: Independent programs access and communicate exclusively through a global data repository, known as a blackboard.

Components: Independent programs, sometimes referred to as "knowledge sources," blackboard.

Connectors: Access to the blackboard may be by direct memory reference, or can be through a procedure call or a database query.

Data elements: Data stored in the blackboard.

Topology: Star topology, with the blackboard at the center.
Variants: In one version of the style, programs poll the blackboard to determine if any values of interest have changed; in another version, a blackboard manager notifies interested components of an update to the blackboard.

Qualities yielded: Complete solution strategies to complex problems do not have to be preplanned. Evolving views of the data/problem determine the strategies that are adopted.

Typical uses: Heuristic problem solving in artificial intelligence applications.

Cautions: When a well-structured solution strategy is available; when interactions between the independent programs require complex regulation; when representation of the data on the blackboard is subject to frequent change (requiring propagating changes to all the participating components).

Relations to programming languages or environments: Versions of the blackboard style that allow concurrency between the constituent programs require concurrency primitives for managing the shared blackboard.

Style: Rule-Based/Expert System

Summary: Inference engine parses user input and determines whether it is a fact/rule or a query. If it is a fact/rule, it adds this entry to the knowledge base. Otherwise, it queries the knowledge base for applicable rules and attempts to resolve the query.

Components: User interface, inference engine, knowledge base.

Connectors: Components are tightly interconnected, with direct procedure calls and/or shared data access.

Data Elements: Facts and queries.

Topology: Tightly coupled three-tier (direct connection of user interface, inference engine, and knowledge base).

Qualities yielded: Behavior of the application can be easily modified through dynamic addition or deletion of rules from the knowledge base. Small systems can be quickly prototyped. Thus useful for iteratively exploring problems whose general solution approach is unclear.

Typical uses: When the problem can be understood as matter of repeatedly resolving a set of predicates.

Cautions: When a large number of rules are involved, understanding the interactions between multiple rules affected by the same facts can become very difficult. Understanding the logical basis for a computed result can be as important as the result itself.

Relations to programming languages or environments: Prolog is a common language for building rule-based systems.

Style: Interpreter

Summary: Interpreter parses and executes input commands, updating the state maintained by the interpreter.

Components: Command interpreter, program/interpreter state, user interface.

Connectors: Typically the command interpreter, user interface, and state are very closely bound with direct procedure calls and shared state.

Data elements: Commands.

Topology: Tightly coupled three-tier; state can be separated from the interpreter.

Qualities yielded: Highly dynamic behavior possible, where the set of commands is dynamically modified. System architecture may remain constant while new capabilities are created based upon existing primitives.

Typical uses: Superb for end-user programmability; supports dynamically changing set of capabilities.

Cautions: When fast processing is needed (it takes longer to execute interpreted code than executable code); memory management may be an issue, especially when multiple interpreters are invoked simultaneously.

Relations to programming languages or environments: Lisp and Scheme are interpretive languages, and sometimes used when building other interpreters; Word/Excel macros.

Style: Mobile Code

Summary: Code moves to be interpreted on another host; depending on the variant, state does also.

Components: Execution dock, which handles receipt and deployment of code and state; code compiler/interpreter.

Connectors: Network protocols and elements for packaging code and data for transmission.

Data elements: Representations of code as data; program state; data.

Topology: Network.

Variants: Code-on-demand, remote evaluation, and mobile agent.

Qualities yielded: Dynamic adaptability. Takes advantage of the aggregate computing power of available hosts; increased dependability through provision of migration to new hosts.

Typical uses: When processing large data sets in distributed locations, it becomes more efficient to have the code move to the location of these large data sets; when it is desirous to dynamically customize a local processing node through inclusion of external code.

Style: Publish-Subscribe

Summary: Subscribers register/deregister to receive specific messages or specific content. Publishers maintain a subscription list and broadcast messages to subscribers either synchronously or asynchronously.

Components: Publishers, subscribers, proxies for managing distribution.

Connectors: Procedure calls may be used within programs, more typically a network protocol is required. Content-based subscription requires sophisticated connectors.

Data elements: Subscriptions, notifications, published information.

Topology: Subscribers connect to publishers either directly or may receive notifications via a network protocol from intermediaries.

Variants: Specific uses of the style may require particular steps for subscribing and unsubscribing. Support for complex matching of subscription interests and available information may be provided and be performed by intermediaries.

Qualities yielded: Highly efficient one-way dissemination of information with very low coupling of components.

Typical uses: News dissemination—whether in the real world or online events. Graphical user interface programming. Multiplayer-network-based games.

Cautions: When the number of subscribers for a single data item is very large a specialized broadcast protocol will likely be necessary.

Relations to programming languages or environments: In large-scale systems support for publish-subscribe is provided by commercial middleware technology.

Style: Event-Based

Summary: Independent components asynchronously emit and receive events communicated over event buses.

Components: Independent, concurrent event generators and/or consumers.

Connectors: Event bus. In variants, more than one may be used.

Data elements: Events—data sent as a first-class entity over the event bus.

Topology: Components communicate with the event-buses, not directly to each other.

Variants: Component communication with the event-bus may either be push or pull based.

Qualities yielded: Highly scalable, easy to evolve, effective for highly distributed, heterogeneous applications.

Typical uses: User interface software, wide-area applications involving independent parties (such as financial markets, logistics, sensor networks).

Cautions: No guarantee if or when a particular event will be processed.

Relations to programming languages or environments: Commercial message-oriented middleware technologies support event-based architectures.

Style: Peer-to-Peer

Summary: State and behavior are distributed among peers that can act as either clients or servers.

Components: Peers—independent components, having their own state and control thread.

Connectors: Network protocols, often custom.

Data elements: Network messages.

Topology: Network (may have redundant connections between peers); can vary arbitrarily and dynamically.

Qualities yielded: Decentralized computing with flow of control and resources distributed among peers. Highly robust in the face of failure of any given node. Scalable in terms of access to resources and computing power.

Typical uses: Where sources of information and operations are distributed and network is ad hoc.

Cautions: When information retrieval is time critical and cannot afford the latency imposed by the protocol. Security–P2P networks must make provision for detecting malicious peers and managing trust in an open environment.

Style: C2

Summary: An indirect invocation style in which independent components communicate exclusively through message routing connectors. Strict rules on connections between components and connectors induce layering.

Components: Independent, potentially concurrent message generators and/or consumers.

Connectors: Message routers that may filter, translate, and broadcast messages of two kinds—notifications and requests.

Data elements: Messages—data sent as first-class entities over the connectors. Notification messages announce changes of state. Request messages request performance of an action.

Topology: Layers of components and connectors, with a defined top and bottom, wherein notifications flow downward and requests upward.

Additional constraints imposed:

- All components and connectors have a defined top and bottom. The top of a component may be attached to the bottom of a single connector and the bottom of a component may be attached to the top of a single connector. No direct component-to-component links are allowed; there is, however, no bound on the number of components or connectors that may be attached to a single connector. When two connectors are attached to each other, it must be from the bottom of one to the top of the other.

- Each component has a top and bottom *domain*. The top domain specifies the set of notifications to which a component may react and the set of requests that the component emits up an architecture. The bottom domain specifies the set of notifications that this component emits down an architecture and the set of requests to which it responds.

- Components may be hierarchically composed, where an entire architecture becomes a single component in another, larger architecture.

- Each component may have its own thread(s) of control.
- There can be no assumption of a shared address space among components.

Qualities yielded:

- Substrate independence: Ease in moving the application to new platforms.
- Applications composable from heterogeneous components running on diverse platforms.
- Support for product lines.
- Ability to program in the model-view-controller style, but with very strong separation between the model and the user interface elements.
- Support for concurrent components.
- Support for network-distributed applications.

Typical uses: Reactive, heterogeneous applications. Applications demanding low-cost adaptability.

Cautions: Event-routing across multiple layers can be inefficient. High overhead for some simple kinds of component interaction.

Relations to programming languages or environments: Programming frameworks are used to facilitate creation of implementations faithful to architectures in the style. Support for Java, C, Ada.

Style: Distributed Objects

Summary: Application functionality broken up into objects (coarse- or fine-grained) that can run on heterogeneous hosts and can be written in heterogeneous programming languages. Objects provide services to other objects through well-defined provided interfaces. Objects invoke methods across host, process, and language boundaries via remote procedure calls (RPCs), generally facilitated by middleware.

Components: Objects (software components exposing services through well-defined provided interfaces).

Connector: Remote procedure calls (remote method invocations).

Data elements: Arguments to methods, return values, and exceptions.

Topology: General graph of objects from callers to callees; in general, required services are not explicitly represented.

Additional constraints imposed: Data passed in remote procedure calls must be serializable. Callers must deal with exceptions that can arise due to network or process faults.

Qualities yielded: Strict separation of interfaces from implementations as well as other qualities of object-oriented systems in general, plus mostly-transparent interoperability across location, platform, and language boundaries.

Typical uses: Creation of distributed software systems composed of components running on different hosts. Integration of software components written in different programming languages or for different platforms.

Cautions: Interactions tend to be mostly synchronous and do not take advantage of the concurrency present in distributed systems. Users wanting services provided by middleware (network, location, and/or language transparency) often find that the middleware induces the distributed objects style on their applications, whether or not it is the best style. Difficulty dealing with streams and high-volume data flows.

Relations to programming languages or environments: Implementations in almost every programming language and environment.

Style: Microservices

Summary: A system that implements the Microservices architectural style consists of a collection of loosely coupled services. Each service is independent, performs a specific function, and communicates with others via asynchronous invocations.

Components: Microservices, each of which is a self-contained service responsible for a specific functionality. Each service exposes API endpoints to interact with other services or clients. There can be a service registry that keeps track of the active services and their locations. There can be a load balancer that distributes traffic across services to ensure optimal performance.

Connector: Asynchronous invocations (e.g., HTTP REST, gRPC, or WebSocket). Message queues to streamline the invocations.

Data elements: Parameters passed along the asynchronous invocations (often in certain data formats such as JSON, XML, Protobuf, etc.)

Topology: A distributed mesh network of services.

Additional constraints imposed: Services must be stateless or maintain minimal state within the service boundary. Emphasis on backward compatibility between versions of APIs.

Qualities yielded: Scalability, resilience (isolated failures), deployment flexibility (independent deployment), modularity (high separation of concerns)

Typical uses: Large, complex enterprise systems. Cloud-native applications. Applications requiring frequent updates and continuous deployment. Systems requiring fine-grained scaling for specific components.

Cautions: High operational overhead, data consistency, network latency

Relations to programming languages or environments: Heterogeneous programming languages and environments. Commonly deployed on cloud.

References

R. N. Taylor, N. Medvidovic, and E. M. Dashofy. “Software Architecture: Foundations, Theory, and Practice,” John Wiley & Sons, 2009. ISBN-10: 0470167742. ISBN-13: 978-0470167748.

Li, Shanshan, et al. "Understanding and addressing quality attributes of microservices architecture: A Systematic literature review." Information and software technology 131 (2021): 106449.