Software Architecture

Jim Fawcett

CSE681 – Software Modeling and Analysis
Fall 2013

Definitions

- "An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces … together with their behavior as specified in the collaborations among those elements, …"[1]
- "… abstract away some information from the system … and yet provide enough information to be a basis for analysis, decision making, and hence risk reduction."[2]
- "…designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives."[3]

References

1. Booch, Rumbaugh, and Jacobson, The UML Modeling Language User Guide, Addison-Wesley, 1999.
2. Bass, Clements, and Kazman. Software Architecture in Practice, Addison-Wesley 1997.
3. Mary Shaw and David Garlan. An introduction to software architecture. In V. Ambriola an G. Tortora, editor, Advances in Software Engineering and Knowledge Engineering, volume I. World Scientific Publishing Company, 1993.

What is Software Architecture?

- The architecture of a software system captures major features and design ideas for a software development project.
  - Describes relationship of users with the system.
  - Describes structure and organizing principles of the system.
    - major partitions within the system and their interfaces
    - responsibilities of, and resources needed by, each partition
    - design concepts: data structures, algorithms, data flows, that help developers understand and implement their piece of the system.
  - Identifies major threads of execution
    - A thread is the sequence of activities that result from some system event. Examples are system startup, response to operator requests, and processing of errors.
  - Identifies critical time-lines and risk areas
    - A time-line is a time-based budget for critical threads.
    - A risk area identifies objectives and requirements that will be difficult to meet under the current architectural and design concept.

Architectural Issues

- Software architecture is concerned with:
  - ***Goals:***
    - main objectives of the system
  - ***Uses:***
    - how people and other software will interact with the system
  - ***Tasks:***
    - activities for system and its major partitions
  - ***Partitions:***
    - packages and objects that make up the system
    - responsibilities
  - ***Interactions:***
    - the relationships and data flows between partitions, and assumptions that partitions have about each other
  - ***Events:***
    - any occurrence that affects system activities
  - ***Views:***
    - appearance of the system to users and its designers
  - ***Performance:***
    - Efficient use of computer resources – processor cycles, network bandwidth, memory

Uses

- Uses describe the way users and other software components interact with the system.
  - What is the user trying to accomplish?
  - What are the required inputs that the user supplies?
  - What are the system outputs that the user expects?
  - What controls will the user want to affect system operation?
- Uses are often developed as scenarios, called use cases.
  - Each scenario describes on or more of the following:
    - User roles, e.g., developer, manager, quality assurance, …
    - Mode of operation, e.g., data collection, data analysis, data presentation.
    - Responses to specific important events, e.g., initialization, user inputs, computational errors, system output.
- Are there effective uses that go beyond the system specification, but would be relatively easy to implement?
  - Can we select a structure that will be easy to extend to these new applications without significant impact on meeting the current requirements?
  - This could result in efficient development of new products and services.
- Impact on design
  - How will the identified uses affect the structure of the system and its design?

Tasks

- Tasks are a high level list of the activities that the system will need to carry out.
  - First developed for the system as a whole.
  - Later, allocated to the major system partitions.

- Tasks are usually presented as lists and in activity diagrams.
  - Activity diagrams are like flow charts, but at a higher level.
  - They describe activities that are important for the system or its major partitions.
  - Activity diagrams show required sequencing and synchronization of tasks.
  - When software is implemented tasks allocated to each package are described in the packages manual page.

Partitions

- Partitions represent the grouping of system activities into logical and physical entities.
  - Package and Module diagrams show the physical packaging of system processing into files.
  - Data Flow Diagrams (DFDs) represent the partitioning of system activities into logical processes, showing the flow of information between the external environment and each process.
  - Classes show the logical partitioning of system data and processing into low-level program constructs.
- Partitions are the second most important part of the architecture concept, after the definition of its tasks.
  - Sequence of concept development is often: (1) uses, (2) tasks, (3) partitions, (4) interactions, (5) events, and (6) views.

Interactions

- Interactions describe the relationships between system partitions. They are described by:
  - ***Data flow diagrams:***
    Used in the early phases of architecture and requirements development.
  - ***Package diagrams:***
    Describe static relationships between the system's physical partitions.
  - ***Class diagrams:***
    Describe the static relationships between the system's logical components.
  - ***Event trace diagrams:***
    Show the dynamic relationships between system components.
  - ***Structure charts:***
    Describes the relationships between the system's functions.
  - ***State charts:***
    Describe the dynamic relationships between the system's computations.

Events

- Events describe specific occurrences to which the system must respond, or that affect its modes of operation.
  - Events are critically important for real-time systems, e.g., systems that must respond to asynchronous events from the outside environment.
  - For these systems, architecture development may revolve around the definition of critical threads.
    - A thread, as defined by the architecture concept, is all the processing that results from a specific event, e.g., a radar detection, user input, power on, computational error, etc.
    - Many threads are defined, then sorted by importance, relative to the system requirements. The architecture isn't complete until processing is defined that will support system requirements for each of the critical threads.
  - Threads are usually described by activity and/or event trace diagrams.
  - In some non-real-time systems events play only a minor role in developing the system architecture.

Views

- Views are used in two ways:
  - Views describe the user interface as it appears to the user.
    - Layouts of controls and screens.
    - Screen shots showing what the user will see when entering data.
    - Screen shots showing what the user will see when observing operation.
    - Each of these views is accompanied with text describing how the user interacts with the controls and screens.
  - Views also describe the most important data structures and algorithms as they appear to the developer:
    - Data structure diagrams are ad hoc diagrams that show how data elements relate to each other.
    - Data structures may be described with xml tree views.

Performance

- Level of communication affects performance by orders of magnitude:
  - Within a process
  - Between local processes on a single machine
  - Between machines in a network
  - Between networks, e.g., across the internet
- Lazy communication:
  - Send information only when needed
  - Send only the specific information needed
- Data caching:
  - Store information locally so that it need not be requested repeatedly
- Minimize remote connectivity:
  - Connections consume threads, CPU cycles, memory
  - Make connection time least necessary to complete request, then disconnect.

Analysis

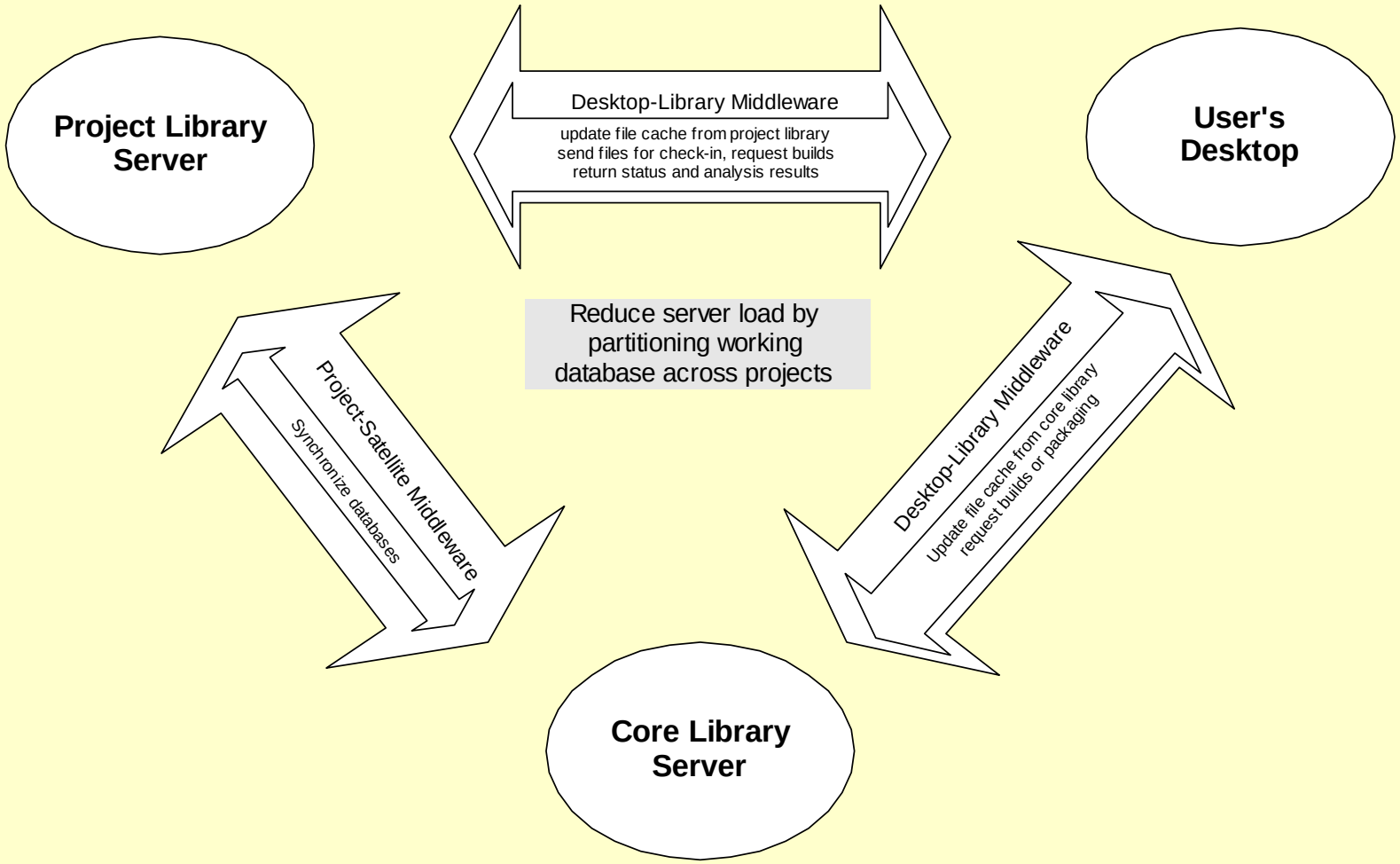
- Analysis of the architecture of a software system entails:
  - Analysis of scale
    - How many users, files, storage size, working set size?
  - Analysis of load
    - Number of concurrent users, open files, open connections
    - Peak and average data flows between process, machines, networks
  - Analysis of timelines
    - How long to initialize and perform key operations?
  - Analysis of function
    - What tasks and operations are essential?
    - How should they be organized?
      - Logical organization is easier to understand, develop, and maintain
      - Data flow often dominates performance
  - Analysis of risk
    - High risk tasks and operations
  - Means of risk abatement
    - What steps can we take to minimize the impact of risk areas?

Software Repository Example

- Background:
  - It is common for software systems to require millions of lines of code for their implementation.
  - The implications of that size and complexity are:
    - Large teams are required in order to complete development in reasonable time.
    - Maintaining conceptual integrity and managing development are extraordinarily difficult.

- Goals:
  - Support massive reuse of existing software so that a large fraction of the total is reused without modification. Issues are:
    - ***Efficient search*** – find a few components out of a collection of many thousands.
    - ***Quality assurance and maintenance*** – support process of maintaining a massive collection of certified components.
    - ***Distribution*** – make components available to all developers anywhere in an organization.

Software Repository Structure - Project and Core Databases



Analysis of Goals

- Assumptions:
  - Quality assurance is primary function of repository server.
    - Reuse implies no change in documentation, code, and test apparatus for the reused components.
    - Very tight quality control is needed for this to be practical.

- Implications:
  - Usability implies that it should be easy to read and extract components.
  - Need for quality guarantees implies that it should be difficult to enter new components and to change existing components.
  - Thus support of analysis tools and role based administration is needed.
  - Support for transition from newly developed component to certified component is needed. A "holding tank" metaphor is appropriate here.

Analysis of Scale

- Assumptions:
  - organization of 2000 developers
  - working on five concurrent projects
  - average software size of 5 million lines of code.
  - ten existing systems fielded and currently operational
  - Each system has a common core of half its total software
  - Average file size is 400 lines of code.
  - Half of the developers are working directly on code at any time.
  - Each developer uses average of five files concurrently.

- Implications:
  - 15 total systems * 2.5 million unique lines of code + 2.5 million lines of common code $\Rightarrow$ 40,000,000 lines of code
  - 100,000 files in repository.
  - 5000 files being used throughout the work day.

Analysis of Load

- Assumptions:
  - One server holding repository software resource.
  - 1000 user's login between 9:30 AM and 11:00 AM, browsing for average of ½ hour each.
- Implications:
  - 500 user hours of service in 1½ hours $\Rightarrow$ 333 simultaneous users
  - This is an untenable load
- Consequences:
  - Partition into system distributed between server and client desktops.
  - Clients browse on their own desktops, make file requests only when needed.
  - Cache files on desktop to minimize server traffic.
  - Use message based communication to minimize length of connection to server.
- Design strategy:
  - Make distributed file management transparent to user.

Analysis of Function

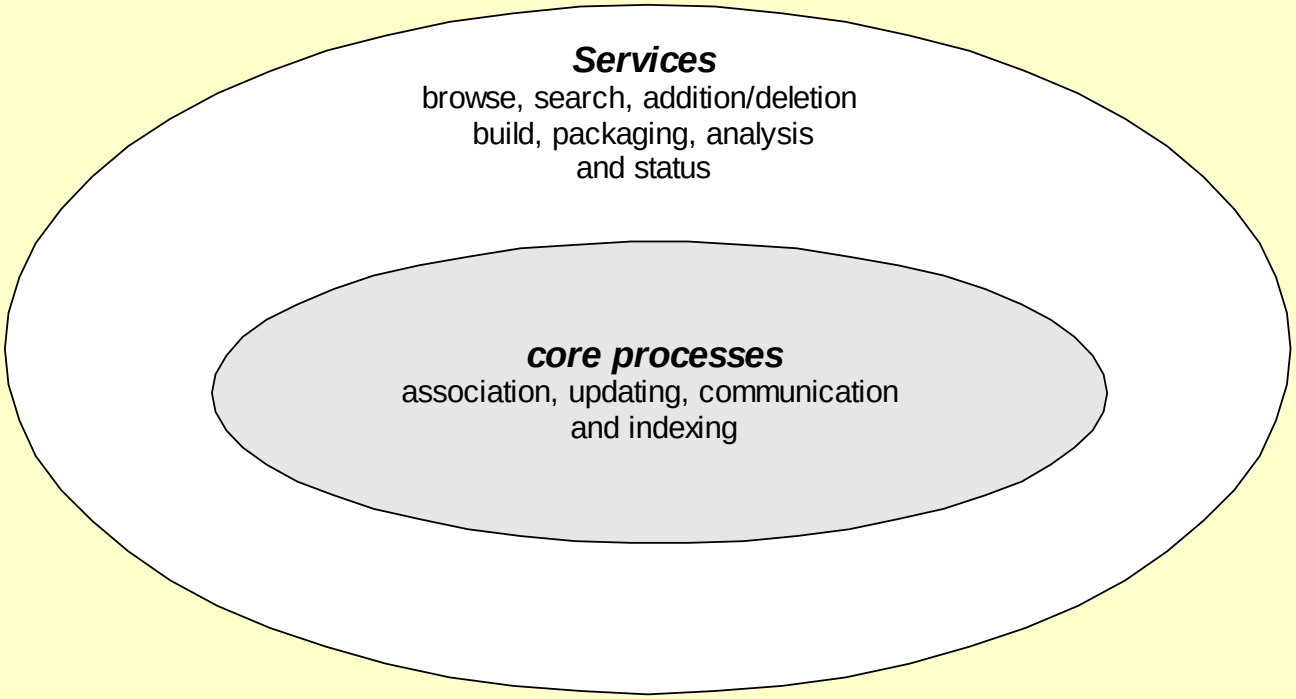
- Need to find a small number of files out of many thousands implies need for:
  - Hierarchical database structure
  - Associations maintained between files, packages, programs, systems
  - Powerful search techniques
  - Support for traversing entire hierarchy on desktop, even though only a small part of the repository's files are cached locally
    - This implies a distributed file management system
- Desire to use repository structure for projects, as well as storage for corporate resources implies need for:
  - Status reporting at every level
  - Simple, but effective configuration management
  - Installable policies
    - Corporate server will make adding new components difficult, e.g., requires tough review process
    - Project servers will make adding new components easy, e.g., requires only that new components have passed unit test.

Summary of Conclusions

Need:

- Hold at least 100,000 components in repository
- Distributed file management system with caching on desktop
- Message passing communication
- Hierarchical associations between components
- Powerful search techniques
- Role-based administration
- Support for quality assurance tools
- Status reporting for all components
- Support configuration management

Software Repository - Services and Processes

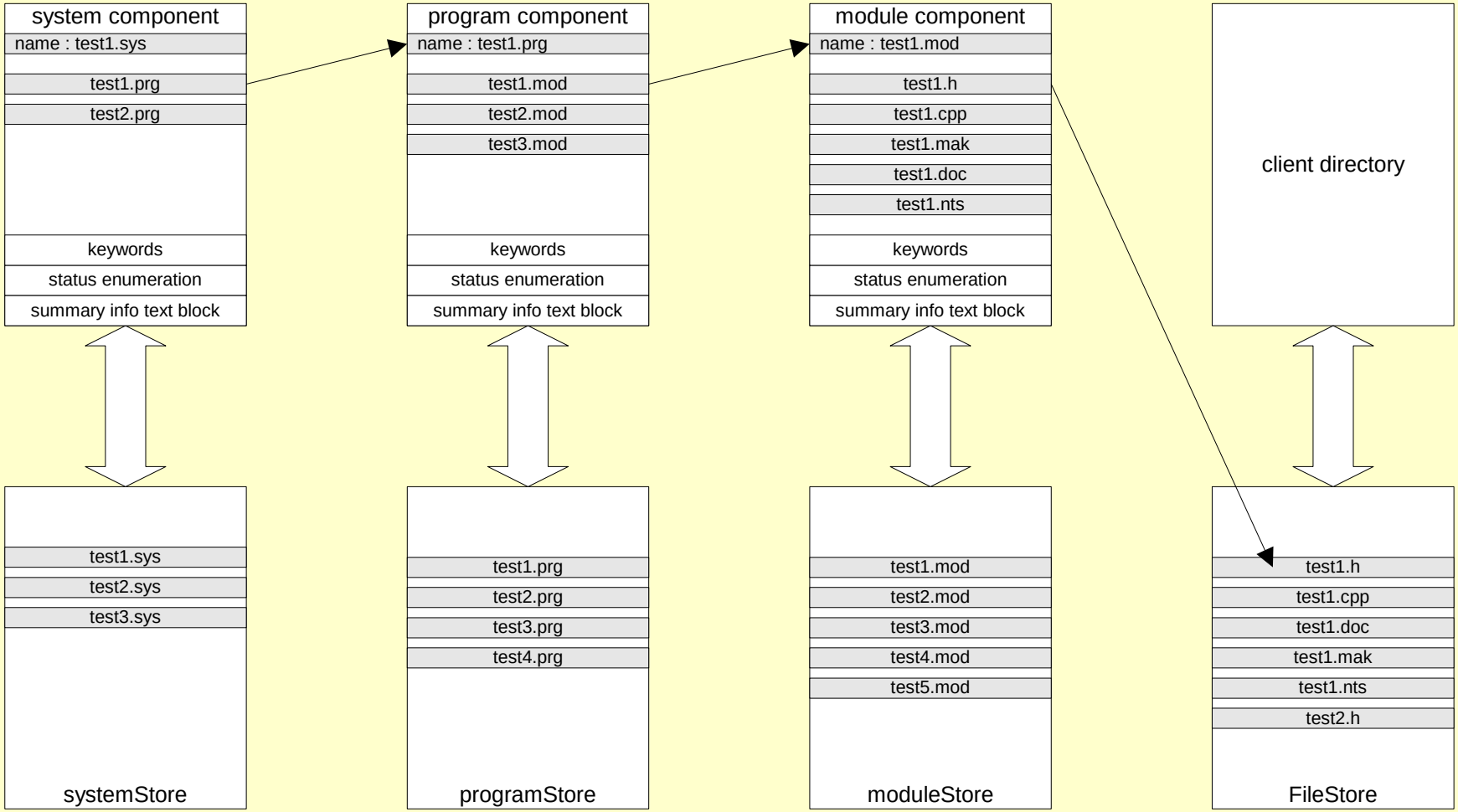


Services are repository processes that the user interacts with

Organizing Principle – Component Structure

- Organize Repository database into:
  - File store: holds exactly one copy of each file
    - Distinct versions are considered to be different files, e.g., token.3.cpp and token.4.cpp are both stored in file store.
  - Package Store, Program Store, System Store
    - Versioned in the same manner as files.
  - Units of reuse are components:
    - Package is a list of files
    - Program is a list of packages and (documentation and test) files
    - System is a list of programs and files
  - Components are represented in the Repository by indexes.
    - An index is a file containing:
      - Path of each lower level component and file
      - Set of keywords
      - Status information
      - Brief statement of function

Components - Defined by Presistent Repository Association Index Structure



Organizing Principle – Search Indexes

- Create package, program, and system indexes to capture the results of searches.
  - Example: Search for thread-safe queue
    - Returns program index pointing to packages that have queue and threading attributes
    - Returns system index pointing to all programs that use these thread-safe queues
  - Example: Program up-date
    - Given some program, search for latest versions of all its lower level components, e.g., packages and files.
    - Return as next version of the program.
    - Obviously, this type of up-date applies to components at any level.

Organizing Principle – File Management

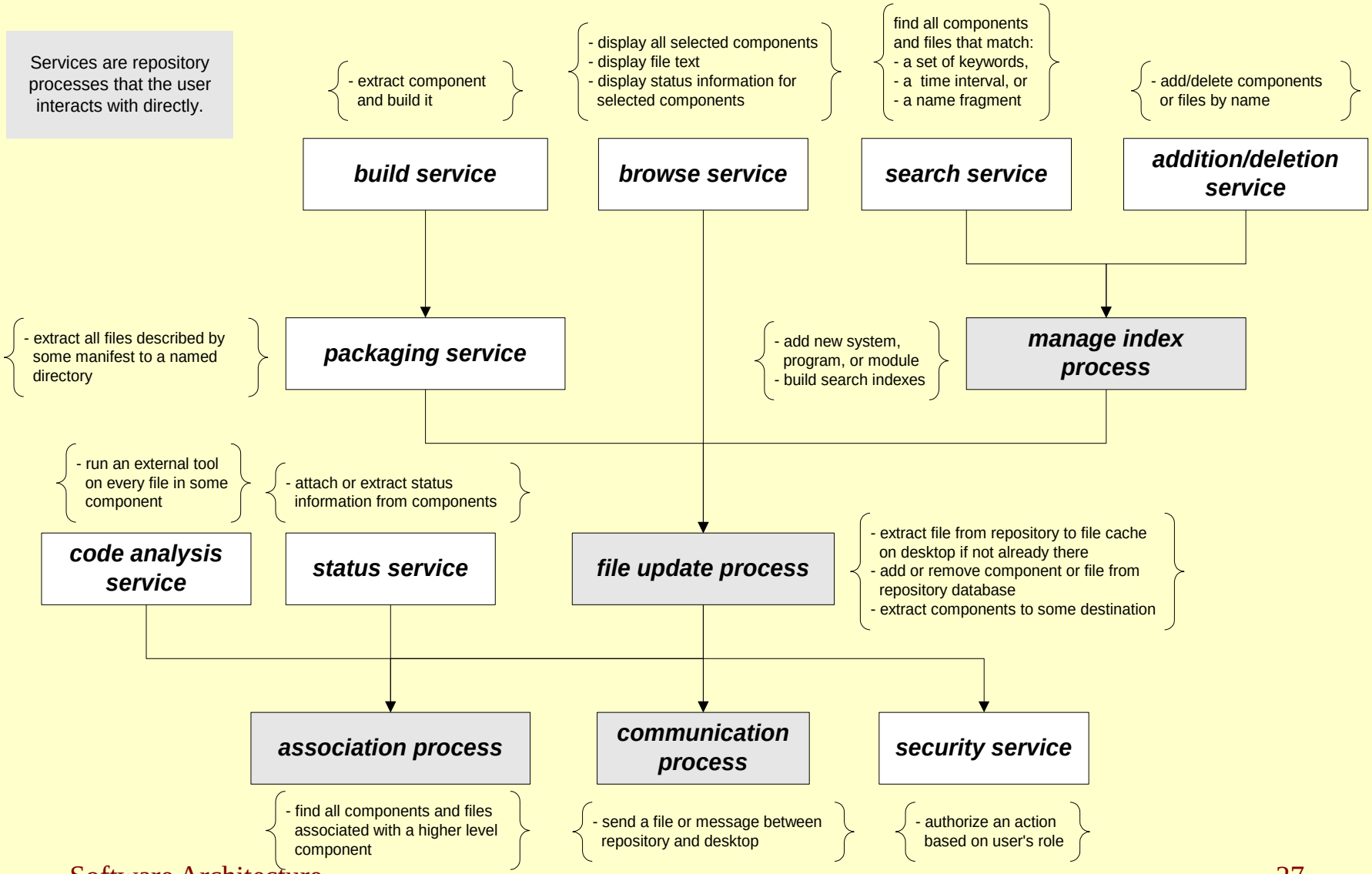
- Operation of Distributed File Management System
  - Every night server sends all its indexes to each desktop.
  - Each index contains keywords, association paths, status information, and a brief text summary of component.
  - Thus client can browse through entire repository structure without making frequent requests of server.
  - Only when client clicks on file link and file is not in local cache will server receive a file request.
  - Client cache is purged only when running low on disk space. Uses least recently used purge algorithm.

Working Set Size

- Estimate of working set:
  - Number of packages = number of files / 4 = 25,000
    - 2 source code files
    - One test driver
    - One documentation file
  - Number of programs = number of packages / 5 = 5000
  - Number of systems
    = 5 current projects + 10 legacy projects + 4 experimental prototypes
    = 19 systems
  - Number of indexes = 25,000 + 5000 + 19 = 30,019 indexes
  - Size of index set = 30,000 * 1 KByte each = 30 MB of index data
- Could greatly reduce this traffic by sending only new indexes each night
  - Occasionally synchronize by sending complete set.

Software Repository - Functional Partitions and Dependencies

Services are repository processes that the user interacts with directly.



Software Repository - Client/Server Structure

Required service time per request implies that most of the repository activities must be conducted on the user's desktop.

The server should be used to serve files and indexes and manage it's internal database

Client Activities:
- manage file and index cache on desktop
- ask server for files and indexes only when cache is out-of-date
- browse through indexes and files
- search through indexes by:
  - date intervals
  - keywords
  - name fragments
  - status and text fragments
- build execution images
- package for deliveray
- collect status

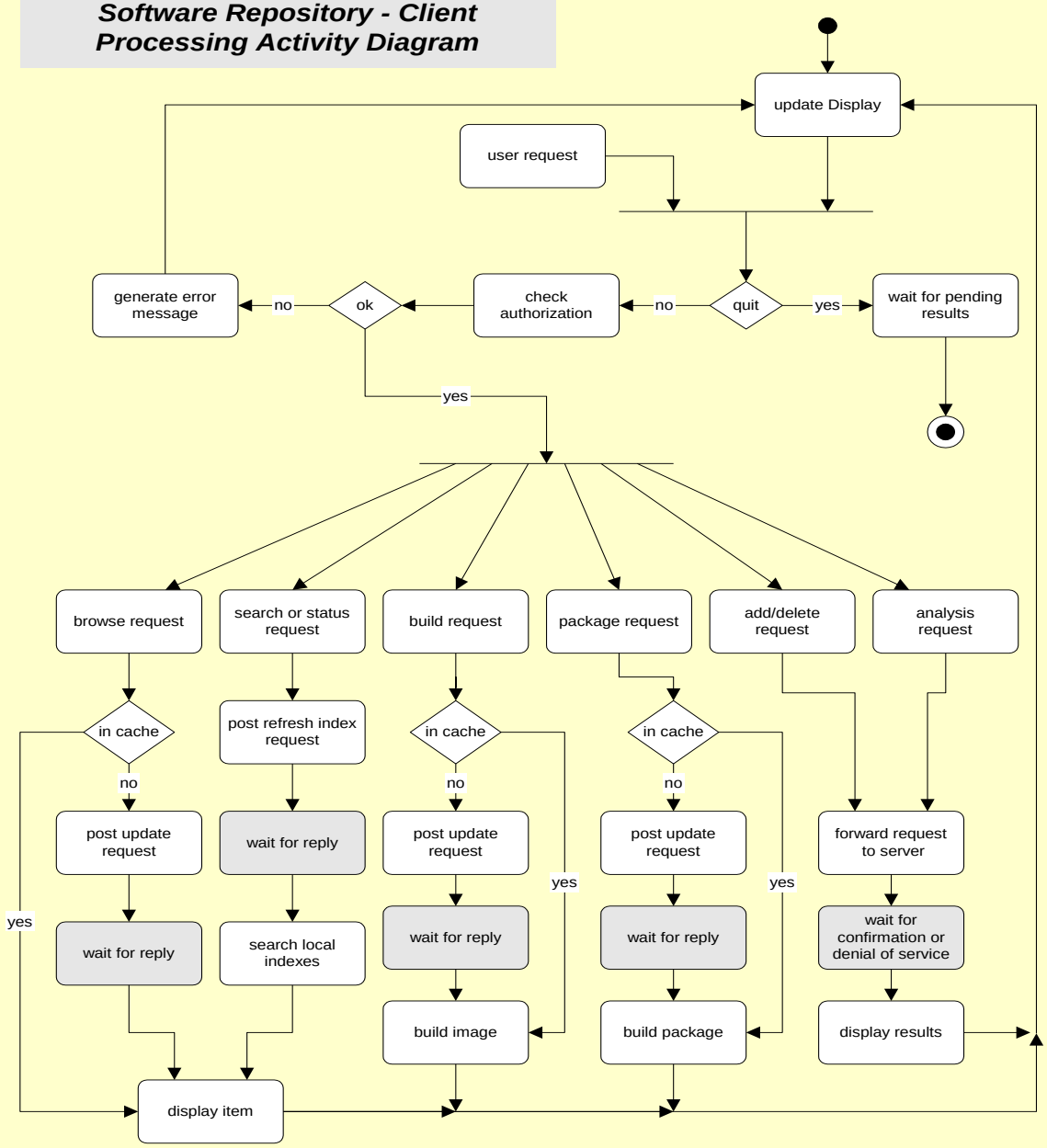
Server Activities:
- check client authorization for requested activity
- serve files to clients on demand
- add/delete database files
- collect status information (off hours)
- package for delivery
- synchronize database contents with other servers

Repository client

client sends requests
server sends back files and confirmations
using communication middleware

Repository server

Software Repository - Client Processing Activity Diagram

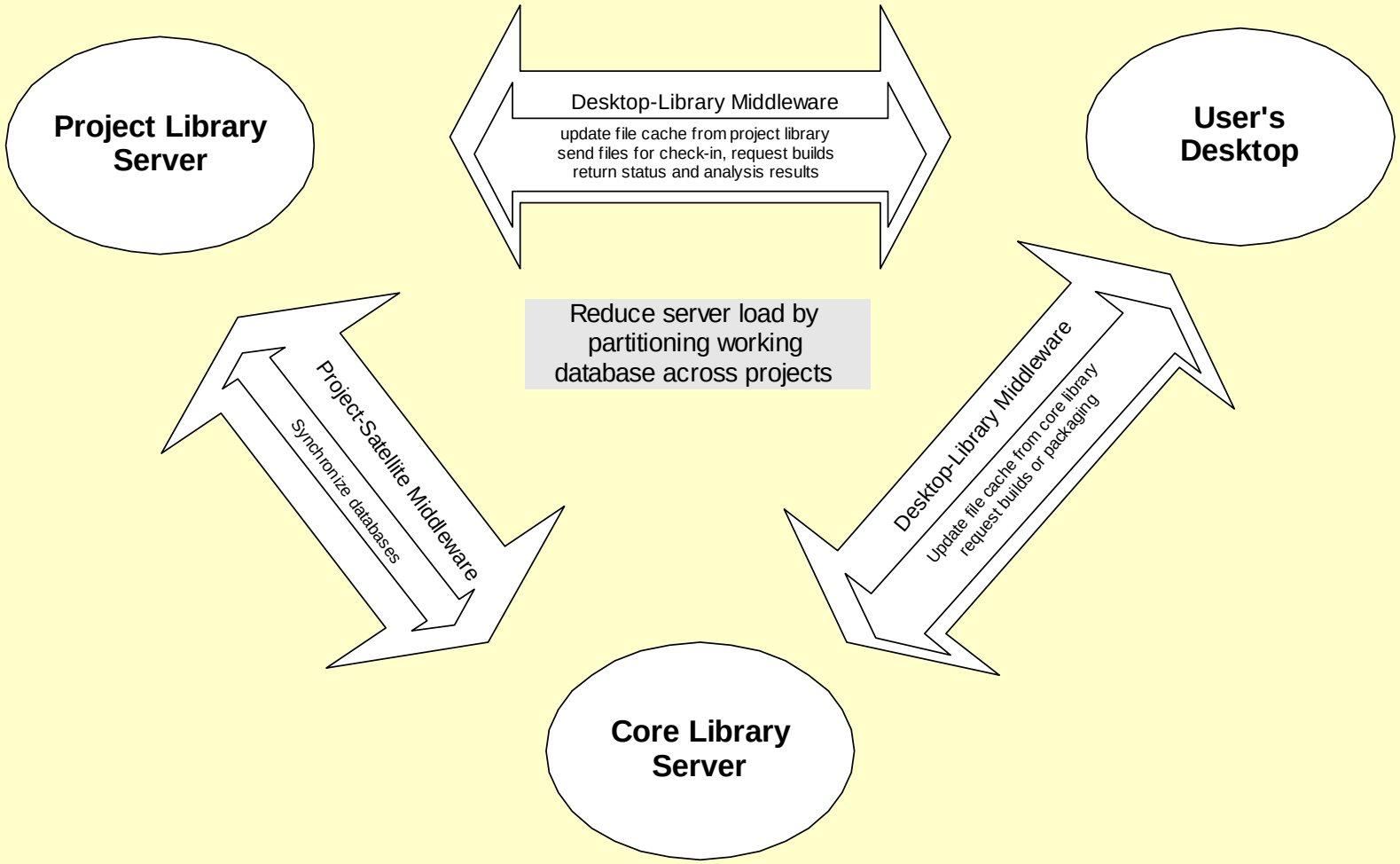


implement with threads so user can initiate other requests while waiting

Organizing Principle – Server Structure

- Organize Repository into a hierarchy of servers and desktops.
  - Primary server holds core repository of reusable components.
  - Project servers are cloned from main repository.
    - Use different rules to manage contents.
      - Easier to add new components and modify existing components.
  - Client desktops are created as subsets of main repository, with contents determined by owner's job functions.
  - Only local browsing is allowed on any server.
    - Only administrators can browse the primary and project servers.
    - Clients only browser the desktop cache.
  - Index distribution supports simulated browsing of the entire repository at client desktops.
    - Distributed file management satisfies file opening requests by first searching the local cache, then downloading from a server if needed.

Software Repository Structure - Project and Core Databases



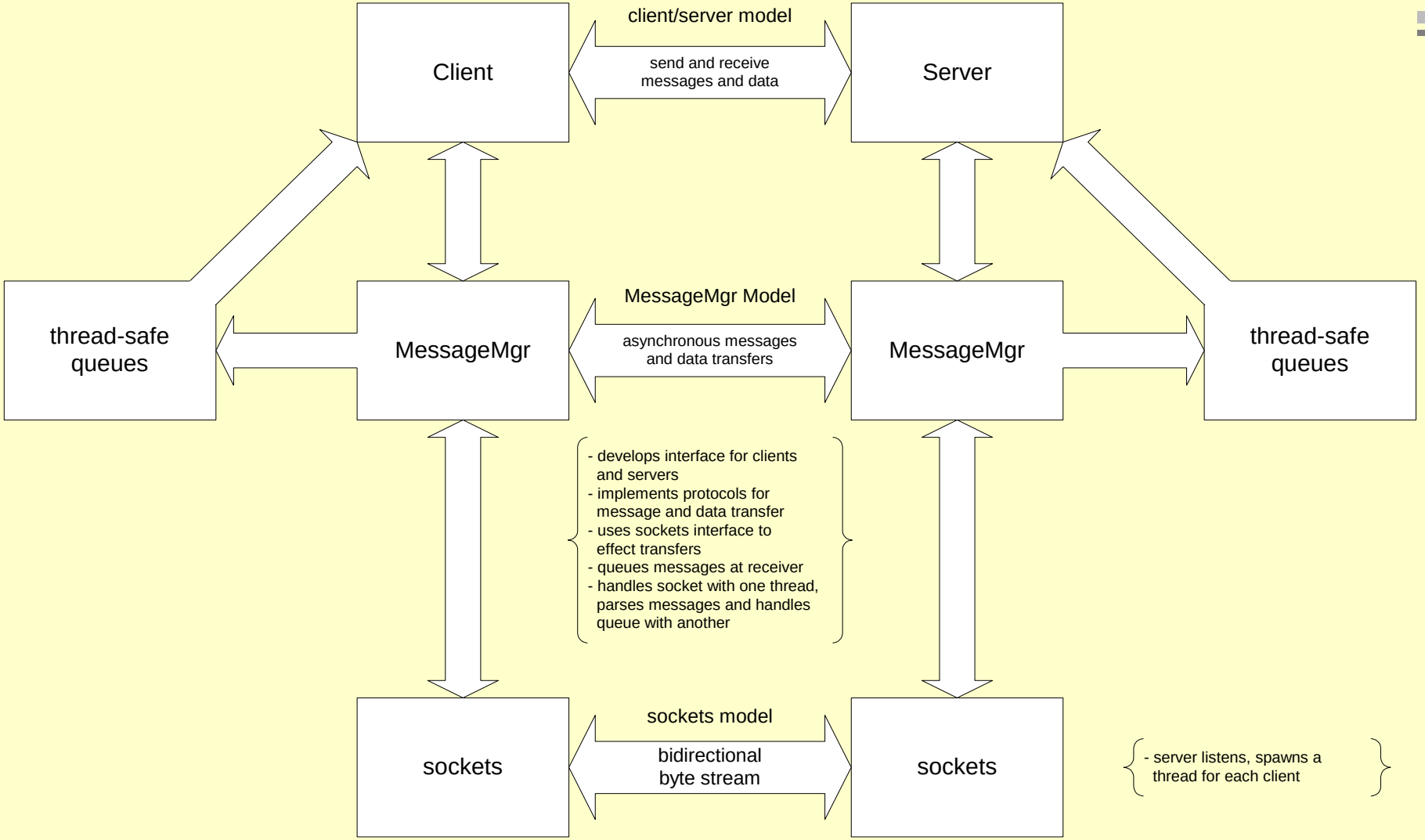
Organizing Principle – Configuration Management

- The proposed repository association index structure can support a very flexible configuration management policy:
  - Every file (source code, document, index) is given a version number, mydoc.doc.3.
  - Indexes refer to a file by name, extension, and version.
  - No file appears more than once in the repository, but all versions are stored.
    - Ancient versions may be retired to an offline archive once no supported system refers to them.
  - Components, e.g., all those objects represented by an index file, are versioned by versioning their indexes.
  - A component can be updated to a new version by updating any one or all of its links to the latest versions of the named files. This automatically increments its version.
  - Since indexes are small files, we can afford to keep all versions under the repository control.
  - Special indexes can be used to point to all the latest changes in the repository, allowing a very flexible updating policy. A team will update to new software, developed by another team, only when they are ready.

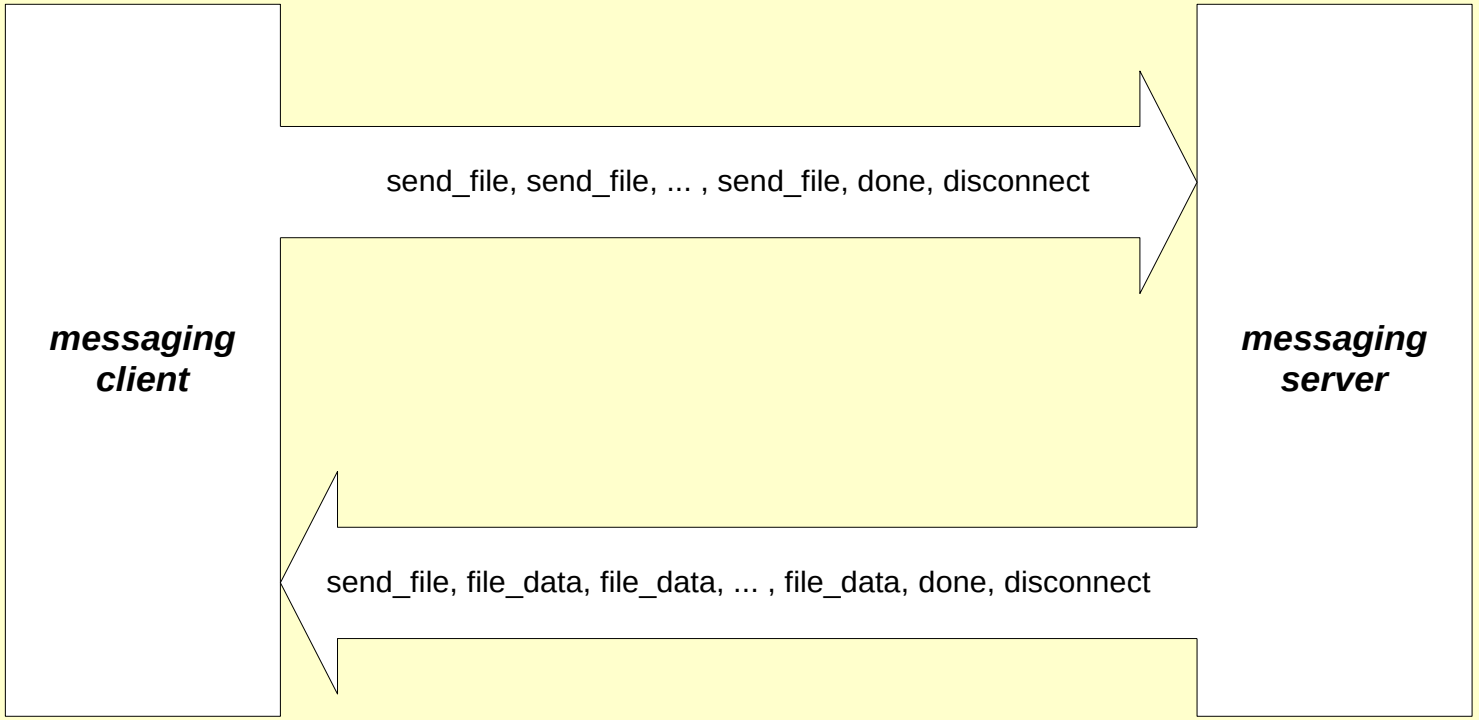
Communication System

- Communications between desktop and servers are based on message-passing.
  - More flexible than Remote Procedure Calls.
  - Client makes request for service:
    - File request
    - Status request
    - Update index request
  - Message is queued on server, serviced based on availability of server CPU cycles and priority – file requests have top priority.
  - Server pushes results to client.
  - Desktop enqueues server results. Queue handler at desktop gets high priority so server does not block.

Project #3 - Messaging System Architecture

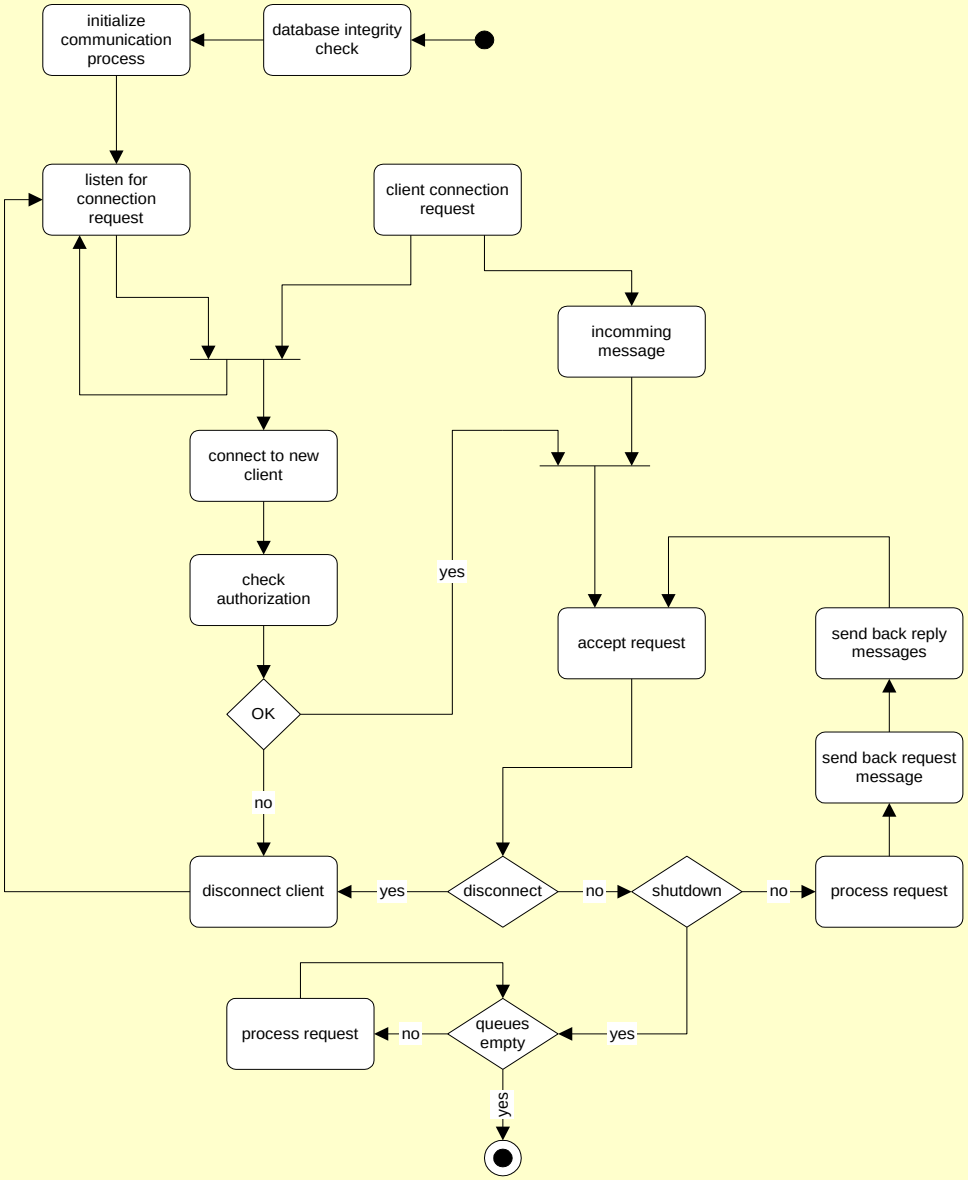


Software Repository - Message Protocol

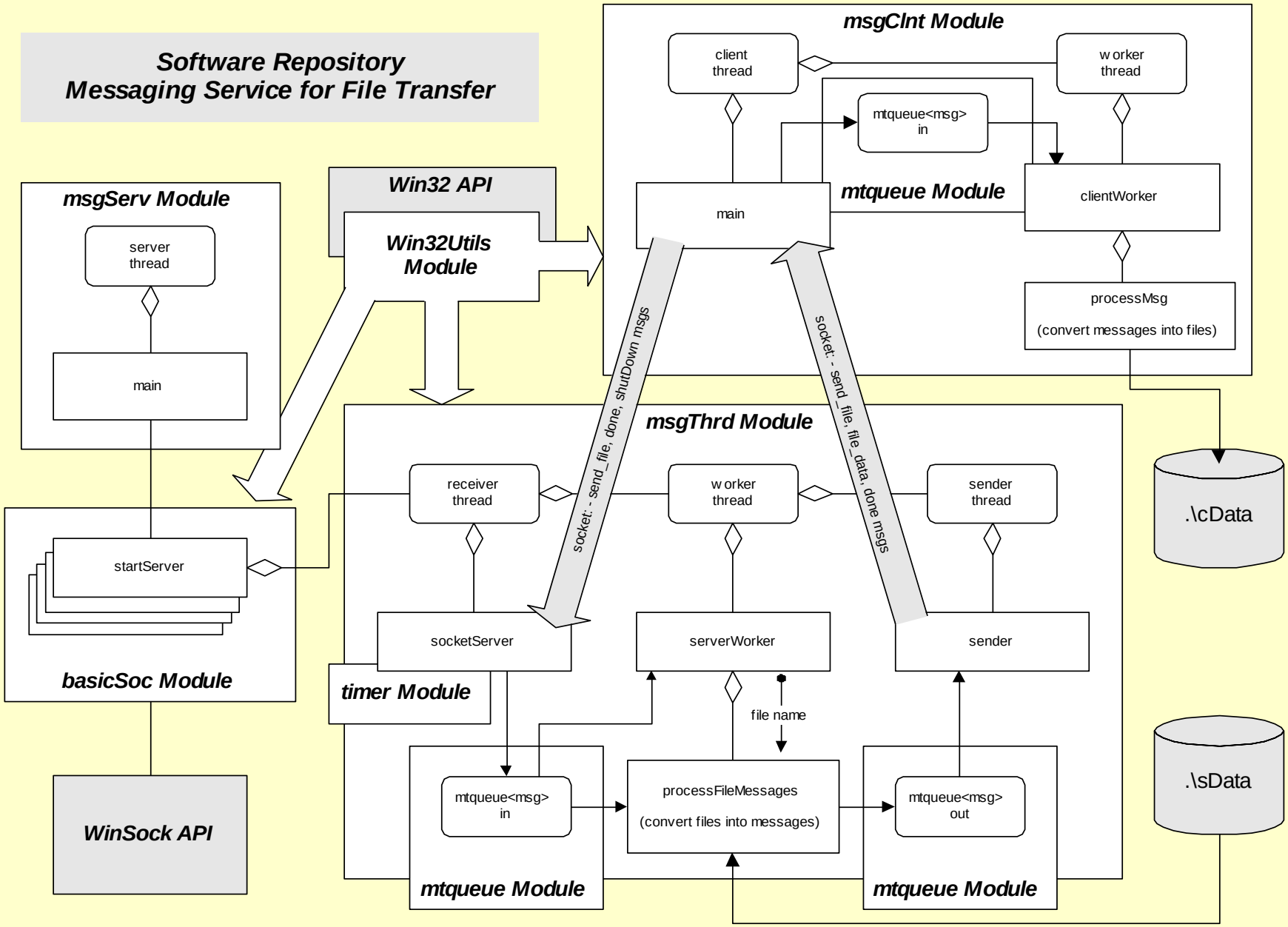


Organizing Principle:
Server reflects back all of the client's request messages and done message

Software Repository - File Serving Activity Diagram



Software Repository
Messaging Service for File Transfer

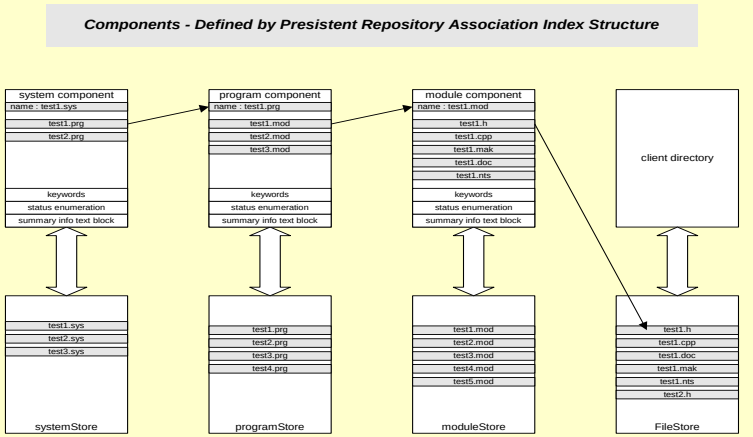


Assessment of Risk

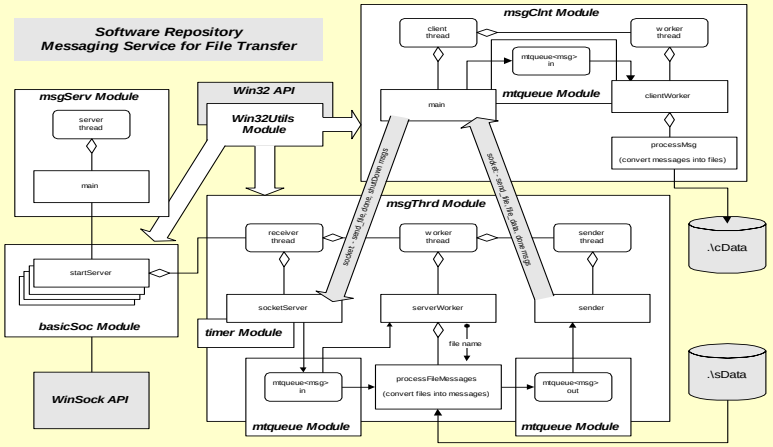
- Risk Areas:
  - Message-passing communication: issues are complexity and performance.
  - Association process, capturing relationships between files, packages, programs, and systems: issues are performance and robustness.
  - Load handling capacity of core repository server: can a single server support demands of a large community of developers?
  - Security: how do we make the repository easily accessible to all our developers, including those at remote sites, while keeping our competitors and malicious hackers from compromising our proprietary software resource?

Risk Abatement

- Association prototype demonstrates that file-based association will support adequate performance. Should investigate other approaches.



- Message passing prototype demonstrates that multi-threaded socket servers support asynchronous file server with required performance.



Risk Abatement

- Repository server performance:
  - Distributed file cache management and index updating service will easily provide adequate performance within a connected network.
  - Should the repository service be provided through web servers, openly accessible world-wide, it will be necessary to provide mirror web hosts. Since the repository contents do not change frequently, this should be easy to manage.
- Repository security:
  - The management plan for the repository calls for anonymous reading, but only administrator write access, conventional security measures should be adequate, e.g.:
    - Firewall controlling the type of access
    - Mirroring contents in multiple sites, including backup sites with no public access.

Risk Abatement

- Protecting proprietary value:
  - Providing strong, conventional network security and allowing outside access only through virtual private networks provide protection against naïve attackers from the outside.
  - Providing read access only through proprietary reader software, different from browser-based access (http and ftp) will add an additional layer of security.
  - There is one gaping security hole in this architecture: the inside job. This system will do nothing to prevent an employee or contractor with repository access from walking out with a box of CDs burned with the repository contents.
  - Two approaches come to mind:
    - Encrypt all contents and decode only on proprietary software keyed to run only on one specific machine, like the Microsoft Activation scheme.
    - Don't treat the repository contents as proprietary.

End of Presentation