Travis Vaughn’s Research/Study Log

06/01/2016 | Word Count: 7200

# Twenty Concepts

## Heavy Server (Thin Client/Thick Server)

A thin client is a client has a smaller amount of code compared to its server. The server does most of the heavy lifting. In this style, updates do not have to be distributed to the clients. Because this style has a heavy server, the program is also much simpler to maintain, with all of the code in one place. The security of the code is increased as it’s not sitting on multiple other clients and finally all exceptions can be handled in one place. Unfortunately this implementation generates significant network traffic, which translates into slow interactive client response. Of course there is an increase in the computation and data burden as it’s all being handled only on the server as opposed to being spread out to clients.

As an example of implementing a heavy server, let’s say I wanted to implement a joke program similar to one of our assignments from the course. If I decided to use a heavy server approach, I may decide to place all of the jokes, along with any type of interaction code (example: how are you, what is your name, etc.) on the server side, along with the state maintenance. Then on the client side, I could leave it bare-bones to where the client code just needs to initiate communication with the joke server. The joke server takes over from there.

Now, in this example, what happens if the joke server goes down or is being used by 100,000 other clients? Well the response time with the joke server decreases dramatically and there’s nothing left for the client to do by sit and wait. However, if we found a bug on the server program, the bug can be easily patched and then the clients will have the updated program the next time they connect. There’s no need to push updates to clients.

## 2. Heavy Client (Thick Client/Thin Server)

A thick client/thin server has most of the code on the client. This implementation reduces the network traffic and allows a quick interactive response time. Exceptions can be handled locally and smoothly. The computation impact of a heavy client is much less than of a heavy server because your able to offload the computation to clients. Without a doubt, the biggest issue facing heavy clients is one of update synchronization. If I’m Adobe and I just released a bug patch to fix a huge security threat, but most of the program code is on the client side, I now have to track the many different clients and push updates to them. What if the clients are offline, you ask? Exactly! Now we have different clients that are out of sync, and we have to wait until their back online.

Using the same joke server example, we could decide to move the jokes and interaction code to the client, and perhaps just have the server maintain the state for each of the clients. In this example if we have 100,000 clients, the response time is still relatively fast as each client is able to generate their jokes on their own machine. We’re just using the server to see where we left off. However if I decide to add new jokes, now I need to push those new jokes to each client machine which will be a huge headache.

## 3. Transactions

Transactions must be atomic, consistent, isolated, and durable. In order to satisfy atomicity, the transaction either completes or fails. Consistency offers that each transaction will adjust from one valid state to another. I should not be able to move the database to some random state that will no other transaction recognizes. Isolation prevents cross-threading issues, specifically by ensuring concurrent transaction executions are handled one right after another. Finally transactions must be durable in that once a transaction as completed, it cannot be undone. This should remain true regardless if the machine crashes or there’s an error.

Transactions were huge topic of discussion during the requirements phase of an application we developed at work. We created a tool that would allow for multiple users (within different functional capacities) to access and update events. To address the atomicity, users saved to complete their process. If the save didn’t complete, then all changes were reverted back to the last save state. All of these transactions could be moved to different states, which we identified as workflow statuses. Certain actions could allow transactions to skip certain workflow statuses, but all transactions went into valid workflow statuses that other actions to access. In order to prevent cross-threading issues, we required users to lock the events being making any updates. This meant while a user was in this event, no other users could modify it; however, they could still work on other events within the tool. Of course once a transaction was saved, no user was allowed to undo it.

## 4. Process

A process is the foundation of work on a computer. It is a piece of code being executed by a computer program. One program can run many processes, but CPUs run processes one at a time, in order; however, this can appear to the user to be running at the same time. Processes user resources such as disk space, printers, displays, etc. On a local system, processes can share program variables through shared locations in memory using what is called inter-process communication (IPC). In distributed systems however, they must send messages, as there is no shared local memory.

Walking through an example, let’s say a computer is running 3 processes: A, B, and C. The user started A at time interval 0, B at time interval 1, and C at time interval 3. The computer will work on A during interval 0, then when B starts up, will begin working on B. Let’s say it finishes B at time interval 2 and then continues working on A at time interval 2 as well. Then at time interval 3, C is initiated and work on A stops. At time interval 4, the computer switches from C to A and finishes A. Finally at time interval 5, the computer continues and finishes C. All of these processes were handled separately, but they were handled so fast that to the user, they appear to happen simultaneously.

## 5. Middleware

Middleware sits between applications and the operating system. It offers a uniform interface that is independent of the OS, which allows greater portability of applications. I might write in C++ and another user might write in Java, but the interface is still the same so the OS can still handle the programs. Examples of this are Java RMI (which is covered extensively later in this study log), and CORBA.

There is a tradeoff between reliability and efficiency. In terms of middleware, think of sending a message and not caring what happens to it afterwards. This is known as send and forget. On the other end of that spectrum you can “Guarantee delivery for every message.” This is likely the least efficient means, but allows for the most reliability to ensure the message gets there.

TCP/IP would be an example stack placing emphasis on reliability over efficiency (but still allows programmer efficiency in allowing XML, a fairly easy markup language to use). UDP is typically associated with other opposite end of the coin: focusing on efficiency over reliability.

## 6. Layered Architecture (Application Layering)

A layered architecture incorporates using many different layers to allocate responsibilities. The component at the top of the hierarchy is able to call the components at the next layer. Control typically flows from layer to layer: requests span down and responses span up. This architecture has become significantly popular in Applications allowing three layers: user-interface, processing, and data. The user-interface level contains everything needed for the user to interact with the application (think GUI). The processing layer would contain the actual applications and the data level is what data is actually being retrieved, stored, or manipulated.

## 7. Event-based Architecture

Event based architectures are usually associated with publish/subscribe systems. At a high-level, processes publish events, which the middleware only allows subscribed processes to access. This is where coupling and decoupling come into play as the big advantage of event-based systems are that they are loosely coupled. Processes are referentially decoupled as they do not need to explicitly refer to each other. Additionally they can be temporally decoupled if they utilize shared data spaces, which means the events and date must be persisted. This is a more complex implementation and is typically associated with cloud computing.

## 8. Forward chaining

Forward chaining is used to implement expert systems. From a logic standpoint, forward chaining keeps repeating modus ponens (P implies Q; P is true, thus Q must be true). Forward chaining is a mode of an inference engine, which is an AI tool to deduce new knowledge. Forward chaining starts with known information and provides new information based on implication (inference). It then continues searching for these implications until it finds one where the If clause is true and then determines the Then clause is accurate, resulting in new data.

## 9. User Datagram Protocol (UDP)

UDP is a simple, efficient way to transmit data across machines. It does not handshake with senders/receivers so there is no way for the sender to know if the receive got the message. This is significant disadvantage in the reliability department. However if reliability isn’t a concern, then this protocol provides significant upticks in speed, as there is no waiting to move to the next transmission. This is useful in time-sensitive applications where losing a packet is better than waiting for delayed packets. One of the popular applications that use UDP is Domain Name System (DNS). This works well for DNS because of the scale of DNS and the burden of reliability is low enough that re-requesting data on failure is of no real consequence.

## 10. Transmission Control Protocol (TCP)

TCP delivers packets across systems utilizing reliability, ordered messaging, and error checking. It is the main protocol of the Internet Protocol (IP) due to its robust offering compared to UDP. For instance, a web server sends an HTML file. TCP divides the file into segments and forwards them individually. When the client receives the packets, TCP reassembles the individual segments in order and ensure they are free of errors. Unfortunately, this protocol can be much slower as packets can be severely delayed or lost altogether, requiring retransmission, thus, it is not desirable for time sensitive applications such as real time systems (think live streaming or VoIP).

## 11. Remote Procedure Call (RPC)

In a remote procedure call, parameters are passed to the remote procedure. The client then waits for a response to be returned from the server. If a response is not received within a certain time frame, the client process can time out. When the server receives the request, it will invoke a subroutine to process the caller’s request, then it will send a reply to the client. Once the client receives this reply, the client process will continue. RPCs are popular with multi-threaded applications as each new request spawns a new thread from the server. The client blocks their thread (waits) until a response is received. The communication protocol between the server and client is created by stubs from the protocol compiler. Users can implement request only RPCs, request/replies RPCs, or request/reply/acknowledge RPCs.

In my line of work, I use this structure when dealing with SWIFT messages. Some messages are informational only, so no reply is needed, others are actions that impact balance orders, so we look for a response from the client acknowledging they received the balancing order, and other times we send information about an event to a client, require a response, and then they want acknowledge that we received their response for legally binding reasons.

## 12. Streams

A simple stream is a sequence, homogenous data made available over time. Think of going to a grocery store and placing items on the conveyor belt and compare it to typing on a keyboard. Putting a can of corn on the conveyor belt is equivalent to hitting a key on the keyboard. Taking the can of corn off the belt at the end is like displaying the typed key on your monitor.

There are also complex streams that consist of two or more sub-streams that are related. For instance, if you have a movie, you will have sub-streams for the video image, audio, subtitles, etc. It can become even more complex when you consider is the data live or stored. How much compression is applied? For instance, longer blocks give better compression, but there’s a trade off as the restarts are longer, so if an error occurs, you’re waiting longer to get going again.

## 13. Quality of Service (QoS)

QoS provides preferential delivery for applications that need it by making sure there is sufficient bandwidth, latency and jitter is controlled, and data loss is reduced. Bandwidth is how much data can be carried over the network at a certain rate. Latency is the delay of receiving data from its source and the jitter is the variation in the delay. For instance, if one transmission takes 0.05 seconds and a second transmission takes 2.5 seconds, the difference is the jitter.

When determining the QoS, you need to decide how urgent the transmission is: are you sending video of a time-sensitive event to the President of the United States where any missing data could impact his next decision? If so, then you need to ensure the highest QoS, which should probably buffer all of the data before beginning those ensuring all packets have arrived and will be displayed in order. Conversely, if you’re looking up a cat picture on your coffee break, it’s likely not as important if you get every pixel of that picture. You’d rather see the bulk of the image before you have to return to work. In this case, you likely need a minimal, to non-existent buffer. If a packet is lost or out-of-order, it will not significantly impact your overall experience.

In the middle of this spectrum is Netflix, which will start buffering in the beginning and then begin the program. If it starts experiencing delays in the packets, it will degrade the quality of the image before it requires a stoppage to buffer.

## 14. Code mobility

This is the ability to migrate running programs from one machine to another. It’s possible to move the code over piece by piece or as a whole. Code mobility can be either Strong or Weak. In strong mobility, the code, data, and execution state are moved from on host to another. Strong mobility is important when the running application needs to maintain its state while it migrates. Weak mobility disregards the state, and only moves the code and the data, which could require restarting the execution of the program on the destination side.

## 15. Super-server

A super server is a daemon that has the ability to start other servers. Resources are minimal when the server is in an idle state. Super-servers are ideal to be used as workstations for client/server development, and internet services. An example of a popular implementation is inetd (internet service daemon). Ientd waits for requests from clients and then spawns a process to run the correct server for the client to handle the connection. The super-servers is typically more memory efficient for lower traffic services since the servers idle when not used. Another benefit is that no network is required in the specific server programs underneath the super-server,

## 16. MIME Types

MIME types are the headers before a message is sent over the Internet that contain certain metadata such as content size, file type, etc. They are useful so that the receiving side knows how to handle the incoming data. More specifically, MIME types consist of a type, subtype, and other optional parameters. In our webserver programs, we had to program to send MIME types for html, xml, and txt documents. For instance, “text/html;” would be the type and subtype. Additionally MIME types can be used to handle newly created file types as long as the linkages are setup correctly, as we saw in our MIMER program (where we used imaginary file type .xyz).

## 17. Network Time Protocol

Network Time Protocol is used for clock synchronization within distributed systems to mitigate the differences in latency of delivery as well as the fact that each machines internal clock can be different. It is important to note that this protocol will not completely synchronize clocks as there can still be inaccuracy at the millisecond level. The clock sync algorithm will periodically test three or more servers on a network. The client must compute the offset of their clock compared to the servers and the round trip delay in order to synchronize their clock.

If a client comes on to the network and begins syncing, it will look at the network’s time, say 12:00:00 UDT, and the client’s time is 11:59:58, then the offset is -00:00:02 and now any communication will include that offset when comparing clocks. Additionally if the time to send to send a package to the network takes 00:00:50, then that amount will be added to the computation so that now the offset and the latency of package delivery are taken into consideration when comparing clocks from system to system.

## 18. Lamport Timestamps

This is an algorithm, named after their create Leslie Lamport, to figure out the order of events within a distributed system. Because different processes are not perfectly synchronized (as discussed in the NTP concept), this algorithm uses what’s call “partial ordering” of events. It is efficient and is a basis for vector clocks (to be discussed in the next concept).

For instance, if I’m trying to access the hard disk with Process 1 and then with Process 2, but there is a delay that causes the processes to arrive at the same time, how is the system supposed to know which one should update first? Lamport’s algorithm solves this issue by applying an incrementing counter that is managed within each process. This counter only matters when messages are moving between processes and when a process receives a message, it synchronizes the clock with the sender.

## 19. Vector Clocks

Vector clocks are also a partial ordering clock synchronization program (similar to Lamport Timestamps). If there are N processes, then there is an array of N logical clocks. Think of the array as the vector. Within each process a local, smallest possible value copy of the global clock-array is kept. All clocks initially start at zero and every time an event occurs in a process, the logical clock/counter is increment in the vector by one. The entire vector (array) is sent along with the message any time a process prepares to send a message. When a process receives a message, it’s local clock in the vector is incremented by one and it updates each of the other elements in the vector by taking the max value of its own vector clock and the value of the vector clock in the received message for every element.

Distributed databases, such as Dynamo, find vector clocks useful because they do not imply more dependencies than actually exist (consider Lamport Timestamps which will apply timestamps to two processes even if unrelated). Instead, these databases separate global clocks by related processes, thus essentially implementing Lamport Timestamps in a vector, where each set of related processes has its own Lamport Timestamp.

## 20. Symmetric Key vs. Public Key Encryption

Symmetric Key is the most common type of encryption (for instance, it’s typically used on passwords when you log into a system). The sender and receiver both use the same key, but the issue is with how to safely distribute the key since both users have access to the same key, but sending and receiving this document is quicker and still allows very good encryption.

Conversely, asymmetric keys allow a solution to the distribution concern raised by symmetric keys. In public key encryption, one key is dedicated to encrypting and a separate key is used for decrypting, but one of them is open, meaning “anybody” can see it. There are certain “oracles”, that are considered to be trusted, that will publish the public key and post the bindings of the public keys to the stakeholders. They essentially certify that the public key is real and that the secret key has not been compromised.

------------------------------------------------------------------------------------------------------------------------------------------

# NEVER MAKE THESE ASSUMPTIONS!!!

1. Network is reliable
2. Network is secure
3. Network is homogenous
4. Topology does not change
5. Latency is zero
6. Bandwidth is infinite
7. Transport cost is zero
8. There is one administrator

# What is a Distributed System?

The system will appear as one system working seamlessly to deliver results; however, behind-the-scenes it is actually a conglomeration of different computers working together.

This means distributed systems:

1. Consist of independently acting agents (ex. Computers)
2. Give appearance that client (human or machine) is only dealing with one system
   1. i.e. the multi-system interactions are hidden from the user
3. Should allow basically the same interaction each time regardless of where and when
   1. Think logging onto a computer in Chicago on a Monday and logging onto a different computer in Dallas on a Friday
4. Scaling should be pretty easy to complete
   1. Due to #3

Because of the 4 items above, distributed systems are often created as a middleware software between the applications and the operating systems. (That’s how it’s able to say “Hey, I see you’re on a Linux box in NYC trying to access this web app ‘Learning to Love Distributed Systems.’ There’s this other person in San Fran accessing it from a Windows computer in order to learn with you and that’s no problem at all.”)

# When should we build Distributed Systems?

We don’t always need distributed systems. There are basically 4 important goals that need to be met in order to build a distributed system.

1. Allows easy access to resources
2. Is transparent
   1. Transparency here means seeming as one system to the client, not showing the user everything we are doing.
3. Is open
   1. It has standard rules to describe syntax/semantics of system
4. Is scalable

## Easy access to resources (to remote resources)

## Resources can be anything: computers, databases, Web sites, printers, etc.

Having easy access to remote resources is economical, provides collaboration (email, WebEx, shared docs, etc.),

Bad news: Security is severely lacking.

* I can buy as many “I Love Distributed Systems” tee-shirts as I want with John Doe’s credit card. There’s no proper identification required.
* Facebook can track all of my information, collect that data, and resell it to others so that now I’m getting a bunch of junk mail. Technically it’s legal in this case since I’ve agreed to their terms and conditions, but not every place is like this.

## Transparency

What are some items the system should hide?

* Access
  + Data representation: Mac OSX file naming convention vs. Windows file name convention
* Location
  + Can’t tell where resource is physically located: URLs. http://www.ilovedistributedsystems.com/index.html does not display location of the main Web Server
* Migration
  + The URL also doesn’t say how long it’s been at its current location or when it moved.
  + This means we can move the physical location the main web server without affecting client access
* Relocation
  + Similar migration, but even better is that a user is currently using the system without any impact while resources are being moved
  + For instance, we have two wireless routers in my home, one for the front of the home and one for the back, when I move from one area to the other, I never notice that I’ve switched connections.
* Replication
  + Replication is the idea of placing a copy of a resource close to the place it’s accessed for performance purposes.
  + *Replication Transparency* deals with preventing the user from knowing several copies exist
  + It’s basically impossible to support replication transparency without also supporting location transparency
* Concurrency
  + Jane Doe accesses a website at the same time John Doe does. They both begin answering questions on the website that is then stored to a database. The system will appear to the users as if they are the only one using it, but technically the system is sometimes locking Jane’s request, until it’s finished processing John’s, but then it may lock John’s other requests while it finishes Jane’s. It happens so fast, that to both Jane and John, it appears like they’re the only ones in the system.
  + The locking allows the shared resource to be in a consistent state.
* Failure
  + A server failing doesn’t impact the client from completing their tasks (and the client has no idea a system has failed)
  + Hiding this is one of most difficult issues in distributed systems
    - Main difficult is telling the difference between a dead resource and extremely slow resource
    - Think about accessing a busy webserver and receiving a timeout request. The user doesn’t know if the server is actually down or not.

When might transparency not be the best solution?

* If transparency affects performance too much
  + Hiding server failure before trying another one
    - May have been quicker to stop earlier and/or let the user cancel
* If replicas need to be consistent at all times
  + Think online gaming.
* May not be obvious that hiding distribution is a good idea
  + I don’t want to print a document at the Lincoln Park campus location when I’m in the Downtown location.

## Openness

Formalized protocols are generally specified through interfaces – typically referred to as an **Interface Definition Language (IDL)**.

Typically only capture the syntax of the service. Thinking functions, parameters, return values, possible exceptions, etc.

Proper Specifications

* Allows process that needs certain interface to talk to another process that provides that interface
* Two different people can build completely different implementations of those interfaces
  + Two different distributed systems that operate the same way
* Are complete and neutral
  + Completeness means everything necessary to implement
  + Neutral means doesn’t describe what implementation should look like
  + These are important for interoperability and portability
    - Interoperability: how two different implementations of systems work together by merely relying on other’s services as specified by the standard.
      * The example I can think of at work is how to access my work’s corporate actions database. There is a service built on top of the database that multiple other services can access (we have trader’s programs accessing, auto posters accessing, etc.). These systems that interact with the corporate actions service also pass along information from the service between each other.
    - Portability: how to move an application developer for system A to system B using same interfaces as A.
* AN IDL CAN BE CREATED WITH A PEN AND PAPER

System should have ability to be configured out of different components and be easy to add new components or replace other components without affecting the components already in place. We call this being **extensible**. For example, adding parts that allows parts to run on different OS.

## Scalability

Three ways scalability can be measured

1. Size
2. Geography
   1. What about when classes become global
3. Admin (often not discussed, but causes some of the most problems)

Bad news: As we scale up in one of these departments, we typically lose performance

Scalability problems

* Size
  + When need more users/resources, confronted with limitations of centralized services (single server for all users), data (single database for all employee records), and algorithms (routing based on complete information).
    - Think services centralized on one server on one machine
    - Sometimes need centralized locations: confidential information
  + Only decentralized algos should be used:
    - No machine has all info on system state
    - Machines make decisions based only on local info.
    - Failure of one machine doesn’t impact others
    - No global clock
      * Saying at exactly 07:00:00 all machines will shutdown will fail because it is impossible to get all clocks exactly synced.
  + Centralization is should be avoided whenever possible
  + Examples: How many students can fit in an online classroom?
    - Bandwidth
    - Room in grading links
* Geography
  + When scaling distributed systems designed for LANs, tough because based on synchronous communication. On WANs, those milliseconds for interprocessing add up over thousands-millions of users.
  + Communication in WANs is unreliable and basically always point-to-point whereas LANs are typically reliable based on broadcasting
    - Example: on a LAN, it can send a message to all connecting services if it’s using service. Only machines that have service respond. This would be awful on a WAN, instead they would need to use special location services
  + Strongly related to problems of centralized solutions
    - Centralized components waste network resources
    - Thinking about playing a game with your buddy online, but every action you take has to first be routed to a server in Hong Kong and come back. The game would be severely lagged.
  + Examples
    - Timing problems: deadlines in the middle of the night for some students
    - Are certain content or components illegal in other areas?
      * Encryption is considered “munitions” (like a gun) in some countries
* Admin
  + Conflicting policies is always a major problem
    - Resource usage and how to pay for it
    - Management
    - Security
  + Examples:
    - How to give 4,000 students grades?
    - Where are assignments stored?
    - Hierarchy of instructors
      * How do they coordinate?

# What are Server Clusters?

## General Organization

Basically it’s a bunch of machines that are connected to one another, where each machine runs at least one server. Specifically this section will discuss server clusters connected over a local-area network (LAN), which offers the benefits of high bandwidth with low latency.

Typically the sever clusters are separated into three tiers:

1. Logical Switch
   1. Client requests come through here and then are dispatched to different application/compute servers.
   2. Example: transport layer switches accept TCP connection requests and pass requests to a server in the cluster.
   3. Important goal is to hide the fact that there are multiple servers from the client, thus providing a single access point.
   4. Switch is point of entry for cluster, offering one network address.
   5. Typically access server cluster via TCP connections where application-level requests are sent.
   6. The switch is necessary for distributing the burden of requests among the different servers, not simply by splitting requests evenly, but also aligning requests most efficiently and effectively with their corresponding server.
2. Application/Compute servers
   1. Use case 1: high-performance machines dedicated to delivering compute power
   2. Use case 2: lower performance machines, think enterprise, the compute power is not the constraint, but access to storage, so higher performance machines are unnecessary.
3. Distributed file/database system

## Distributed Servers

Typically there’s an administrative process managed separately on a different machine. This process is responsible for keeping track of the available servers and passing that information to other machines (like the switch).

Most clusters offer a single access point, so when the access point goes down, the cluster is down as well. In order to avoid this problem, you can adjust the cluster to allow for multiple points of access (example: the DNS can provide multiple addresses that point to the same host), but still creates an issue of not having static access points. Because of the desirability of having a static access point while still having flexibility in configuring a server cluster (including the switch), there was an implementation of a distributed server which is a dynamic set of machines with multiple access points, but rolled up into one access point to the client.

## Managing Server Clusters

Most common approach similar to managing functions of a single computer to that of the cluster. For instance, using administrator rights to connect to a node from a remote client and execute local managing commands to modify, install, and view components. A more complex version is provide an interface on a specific administrative machine that allows collective information gathering of all nodes and ability to push upgrades, and add and remove nodes. Strong advantage is that any operations that impact many of the machines, are more efficiently completed. As soon as nodes increase past a 30 or 40 nodes, this type of management is not feasible (think data centers). Can’t manage these via centralized methods of administration. Support for these types of clusters is typically done differently based on the implementation. While there are common practices to avoid there is no common theme for managing large server clusters.

# What are distributed object-based systems?

## ARCHITECTURE

In the basic sense, distributed objects are objects (think Java Objects) that are transferred between other machines (or processes on the same computer).

In OOP, there is a difference between the interfaces and the objects implementing the interfaces. So when the client wants to interact with the object, we only need to place the interface on the client, meanwhile the object, state, method, and interface can reside on the server. The interface acts as the link between the client and the server. Now this "proxy" interface on the client is loaded into the client's address space, making the machine think the object is actually local (this is known as Location Transparency). This proxy is responsible for marshalling methods into messages and unmarshalling the reply messages from the server to get the result of the method back to the client.

### -Compile Time vs. Runtime Objects-

Compile time = Java, C++ (think classes)

Easier to build distributed applications as programs are easy to maintain and transport from system to system.

Tradeoff is that there's a dependency on that specific language being used.

Run Time = independent of languages (can you multiple languages)

Implementation of objects are left open, so allows flexibility between communications of different programs

### -Example: Enterprise Java Beans-

This is a java specific object that allows different ways for clients to invoke objects hosted by a special server. Application functionality and systems functionality (think looking up objects, storing objects, etc.) are able to be separated by the server.

EJB is housed in a container which allows for readily available access to correct references to other services (JDBC, JNDI, JMS), making the entire binding and use of these services essentially automatic. One caveat is that the programmer needs to clearly delineate between four kinds of EJBs:

1. Stateless session beans
   1. Deals with transient objects: meaning objects are discarded after use
   2. Think a calculator
2. Stateful session beans
   1. Uses short-lived persisted objects: meaning can be called even when not in address space or actively being used, but that eventually it removed...typically after some final process is finished.
   2. Think Amazon shopping cart, continuously adding items, then coming back to review and finally order, then removing object after finished
3. Entity beans
   1. Uses long-lasting persisted objects. Stored in DBs and typically part of distributed transactions.
   2. Think storing a customer’s address for online orders. Object is not removed unless customers specifically request to remove it.
4. Message-driven beans
   1. Objects that react to incoming messages and responding if necessary: this is a publish/subscribe model.

## PROCESSES

### -Object Servers-

Servers specific to supporting distributed objects. As opposed to other servers, object server doesn't actually provide a specific service, rather services are implemented by the OBJECTS that reside within the server.

### -Object Adapter-

Activation policies: Noting that in order to invoke, an object must first be brought into the server's address space, this addresses the question of how to invoke an object.

Object Adapter (wrapper): group activation policies. Think implementation of activation policy.

## COMMUNICATION

Typically uses remote procedure calls (RPCs) to invoke object, but a couple of concerns needs to be addressed beforehand.

### -Bind Client to Object-

Distributed object systems typically provide system wide object references. This means not just between different machines, but also between different processes on the same machine.

When process has an object reference, it must first bind the object before it can use any of its methods. Binding allows a proxy to placed in the address space, allowing the interface to be implemented that has the methods the process can invoke. As mentioned, this binding is typically done automatically.

#### Implementing Object References:

A simple one would include network address of machine where actual object resides along with end point of server managing object, plus indication of which object. Some of this information will be provided by the object adapter. Downsides to this are that if the server's machine crashes, the server has a different end point when restarting, so all object references become invalid.

Typically this is solved by having a local daemon per machine listen to a known end point and track the end point assignments across servers in an end point table, thus when binding a client to an object, we go through the daemon and ask what the server's current end point is. This means we need to give each server an ID in order to index into the end point table.

Issue with this approach is it's impossible for server to move to another machine without destroying all the references to the objects it manages. Solution: Move local daemon to a location server that keeps track of machine where object's server is running. Object reference would have network address of location server, and system ID for server.

All previous info assumes client and server have already agreed to use same protocol, but we can remove this assumption by adding additional information about object reference to include protocol being used.

### -Static vs. Dynamic Remote Method Invocations-

Remote method invocations (RMI) are similar to RPCs with regards to marshaling and parameter passing, however RMIs typically support system wide object references.

Static invocation: using predefined interface definitions implies that if interface changes, client application must recompile before it can use new interface.

Dynamic invocation: Application looks for most recent interface at run time.

### -Parameter Passing-

Passing parameters typically less restricted in RMI systems (because of support system wide object references). Tradeoffs exist between transparency vs. efficiency and difficulty of implementation.

Java RMI: Any primitive or object type can be passed as parameter to RMI as long as it's serializable. Local objects = passed by value; remote objects = passed by reference. So a local object is copied then the copy is used as parameter value whereas the remote object passes the reference as a parameter. In Java, proxies are able thus treated as local objects, and since they are serializable, can be passed as a parameter in an RMI, thus the proxy can be a reference to a remote object. This works in Java only because each process is executing JVM (the same execution environment).

### -Object-Based Messaging-

While RMI is preferred for handling communication (specifically in OO distributed systems), messaging is a compelling alternative, with CORBA being the one of the more well known. Also Java uses it's own Java Messaging Service (JMS).

## SYNCHRONIZATION

Issue: Implementation details are hidden behind interfaces

Occurs when: process invokes a remote object (because it has no idea if it is invoking other objects), could be calling an object that is protected against concurrent access.

Java RMI only allows blocking remote objects on proxies in order to avoid this. Threads in same process CANNOT access same object concurrently. Threads in different processes CAN.

# What is Block-Chaining?

Blockchaining is a distributed database most known for being the underlying technology to bitcoin. It maintains a continuously-growing list of data records that are extremely difficult to tamper with. Each block holds batches of individual transactions and the results of any blockchain executables along with a timestamp and information linking it to the previous block. The cryptography on a blockchain is considered so powerful as to be impossible to break. The blockchain resides on a distributed network of servers rather than a single server and thus whenever a new transaction occurs, the blockchain is authenticated across the network, then included as a new block on the chain. A blockchain consists of two kinds of records: transactions and blocks.

## Transactions

Transactions are the details of the “smart contract” stored in the blockchain. Participants in the system generate transactions anytime a sender sends a contract to a receiver. A valid transaction is defined by the system implementing the block chain, for instance – is it required to be digitally signed, do certain fields need to be required, are certain transactional outputs required to be greater or less than previous transactions, etc.

## Blocks

Blocks are the administrative part of the system, handling the metadata of the transactions. They confirm when the transactions occurred and what order the transactions are entered and logged into the block chain. In cryptocurrency systems, “miners” are paid fees from transactions and defined awards for each block produced that are paid to whichever miner successfully confirms the transaction.

## Decentralization

Because every node in the system has a copy of the blockchain, it’s unnecessary to have a trusted third party system manage the process. Transactions are broadcasted to the network using applications and network nodes can validate transactions, add them to their copy and then broadcast the additions to other nodes. In order to avoid needing a third party to provide timestamps on transactions, decentralized blockchains use various timestamping schemes.

## Advantages

1. Independent nodes come to an agreement on the latest version of a ledger, even if there are malicious, inaccurate, or anonymous users or poor connectivity.
2. A node that is well-connected can determine whether a transaction does or does not exist within the data set with a resounding certainty.
3. After the confirmation of a transaction created from a node, the node has certainty over whether the transaction is valid and can become finalized. This is necessary to ensure no other conflicts occurred from submission to confirmation, that no other details were added after the initial transaction was generated, and/or that there is no duplication of the same transaction.
4. The cost to try and modify previous transaction is too high to be considered a threat.
5. Automated conflict resolution – if two or more transactions conflict, they will never be added to the confirmed data set.

## Impact

Blockchaining has significant use with public ledges and is being researched for use by firms in the financial industry, with J.P. Morgan leading the way. If implementing this new technology could replace up to 50% of middle and back office employees that are responsible for manually verifying settlement details on transactions. With blockchaning, there is no need to verify these details manually as the same message is agreed to by both executing counterparties, thus no need for differing settlement confirms. Additionally all transactions would be open to market participants allowing for greater transparency in a typically secretive industry. There has been significant discussion around whether execution traders are truly getting the best prices for their customers, rather than exploiting price fluctuations in the market to pad their spread on transactions.

In short, Russian ex-minister of Economics compares the impact of blockchain technology with the impact of the creation of the Internet in the banking operations realm.