Architecture Patterns vs Quality Attributes & Tactics

Chen Junda

161250010

Software System Design and Architecture

January 8, 2019

Availability

Pattern	Detection			Recovery							Reintroduction	
	Ping-Echo	Heartbeat	Exception	Redundancy	Voting	Spare	Removal from Service	Transaction	Process monitor	Rollback	State Resync	Shadow Operation
Layered			X					X		X		X
Broker	X	X	X	X	X	X	X	X	X	X	X	X
MVC			X	X					X	X	X	X
Pipe-filter			X									
Client-server	X	X	X	X	X	X	X	X				
P2P				X	X	X	X			X	X	
Service-oriented			X	X	X	X	X	X	X	X	X	X
Publish-subscribe				X							X	
Shared-data	X		X							X		
Multi-tier	X	X	X	X	X	X	X	X		X	X	X
Map-reduce	X	X	X	X		X	X	X				

Pattern	Benefits	Penalties
Layered	Each layer is mostly isolated so that failure inside one layer won't affect others and can be fixed individually.	A complete failure of one layer (i.e. all instances are down) will disable the whole system.
Broker	The complete separation of client and server makes tactics to detect and recovery from fault for client and server much easier to implement. The broker can also act like a monitor to improve server availability.	The broker can be a single point of failure thus the failure of broker alone will disable the whole system.
MVC	Models and views can provide redundancy to improve availability.	All three parts are tightly coupled so any of them is single point of failure.
Pipe-filter	Each filter is isolated so that it can be easily replaced during recovery.	Filters are tightly connected so that no redundancy can be achieved, and failure of any filter will shut down the whole system.
Client-server	C-S behaves like broker with broker included in the client so that it shares most benefits with broker pattern.	The server is a single point of failure; since the connector usually doesn't care about fault recovery in server, the server must deploy its own tactics to maintain and restore service during failure.
P2P	Failure of some peers in a big P2P network won't affect the system because of the huge redundancy so the possibility of such a system to shutdown is extremely low.	A small P2P network may not be as consistently behaving as a big network.

Service-oriented	SOA behaves like a complicated broker pattern (if server is called intermediary) or a complicated C-S pattern (if directly) so they share the same benefits with C-S ad broker.	The orchestration server or ESB can be a single point of failure if used; the server might be the single point of failure if called directly.
Publish-subscribe	Redundancy can be introduced for each component and the failure of one component won't spread.	It's hard to detect failure of a component and therefore hard to recover from failure.
Shared-data	Data stores can perform checkpoints and rollback when failure to restore from failure.	Data store is a serious single point of failure; the failure of data store will affect all related services and even the whole system.
Multi-tier	Each tier can be separated physically so that each tier can use tactics to ensure the availability of each tier which improves the overall system as a result; also prevents failure of a tier from spreading to others.	No substantial penalties to availability; the cost of building such a tier is too high.
Map-reduce	The statelessness nature of map instances and minimal connections between modules enables quick and easy recovery from failure.	The use cases map-reduce applies to is limited.

Interoperability

Patterns	Locate	Manage Interfaces	
	Discover Service	Orchestrate	Tailor Interface
Layered	X	X	X
Broker	X	X	X
MVC		X	X
Pipe-filter	X	X	
Client-server	X	X	
P2P	X		
Service-oriented	X	X	X
Publish-subscribe	X	X	X
Shared-data	X	X	X
Multi-tier	X	X	X
Map-reduce	X	X	

Pattern	Benefits	Penalties
Layered	Each layer only calls interfaces from the lower layer and exposes interfaces to the upper, enabling locating and managing interfaces easy.	The use of interfaces is limited on the platform it defines in; the interfaces a layer exposes should be complete and stable, increasing design cost.
Broker	The broker and proxies manage interoperation from other systems, making both ends much easier to implement. The change in one end doesn't have to notify another end.	Broker must handle all interoperability, increasing its design cost; the interface change of broker must notify both ends to adapt.
MVC	Controller can orchestrate complicated requests.	An MVC system is not designed to be interoperated; It doesn't have to expose good APIs.
Pipe-filter	Each pipe focuses on one specific task and only relies on well-formed data, so it is easy to interoperate.	The task of each pipe is predefined and hard to change.

Client-server	Technically the services can be acquired from any clients since the interface is independent from clients.	The change of service in server must notify all clients.
P2P	Any peer can easily acquire service by joining the network.	The uncontrollability to peers makes changing interfaces nearly impossible.
Service-oriented	It is designed to be interoperated and provides a service registry, orchestration server specifically to assist in the process.	No substantial penalties to interoperability: this is designed to be interoperated as mentioned in the left.
Publish-subscribe	The manager or connector or event distributor manages event announcement and listening.	The event distribution process might be limited on a specific platform or technology stack.
Shared-data	Only data store should be accessed to interoperate with the system since it delegates all communications.	The change of any service or the data store must notify the other end.
Multi-tier	Each tier only calls interfaces from the next tier and exposes interfaces to the previous, enabling locating and managing interfaces easy.	The interfaces each layer exposes should be complete and stable, increasing design and cost.
Map-reduce	The task of a map-reduce system works as long as well-formed data is provided, so it is easy to be interoperated by other services.	The task of a such system is predefined and hard to change; the input must be able to be divided into similar subsets, limiting its use cases.

Performance

Pattern	Control Resource Demand				Manage Resources			
	Manage Sampling Rate	Limit Event Response	Prioritize Events	Reduce Overhead	Increase Resources	Concurrency	Multiple Copies of computation	Multiple Copies of Data
Layered								X
Broker	X	X	X			X	X	X
MVC			X			X		
Pipe-filter						X		
Client-server	X	X		X				X
P2P					X	X	X	
Service-oriented	X	X						X
Publish-subscribe		X	X			X	X	
Shared-data								X
Multi-tier								X
Map-reduce					X	X	X	

Pattern	Benefits	Penalties
Layered	Cache can be made in each layer.	Layered lowers performance by introducing overhead and indirection.
Broker	Broker can prioritize and queue events, introduce concurrency by allocating tasks to different servers or introducing cache.	Broker is introducing indirection and bottleneck thus adds latency and reduces performance.

MVC	MVC can prioritize and concurrently run different type of work (like user interface to the top priority) accordingly to maximize performance and user experiences.	The communication between components can be costly sometimes.
Pipe-filter	A work might be divided into multiple parts to distribute to different pipes to run concurrently.	The interpretation of information between filters can reduce performance significantly if the number of communications is large.
Client-server	The connector can specify cache strategy; The direct connection between both sides can reduce overhead.	The communication between clients and server can be expensive if a protocol is not specifically designed for performance.
P2P	The workload can be distributed to peers to improve performance.	Transferring data among many peers can be a huge performance overhead.
Service-oriented	The connector can specify cache strategy to improve performance.	Service discovery and acquirement can be costly. Middleware can be performance bottleneck.
Publish-subscribe	The event subscribing reduces unnecessary data transfer and enables concurrency task prioritization and distribution.	The distribution increases latency and reduces predictability on the response time.
Shared-data	The shared data can set up cache strategy.	The shared data store can be a serious performance bottleneck under concurrent tasks.
Multi-tier	Cache can be made in tiers.	The communication and indirection between tiers can be a performance bottleneck.
Map-reduce	Big amount of data is divided into multiple subsets and run parallelly, significantly reduces time and improves performance.	The data interpretation, division and reducing can be bottleneck.

* Most patterns introduce (instead of reducing) overheads in exchange for other attributes.

** Patterns doesn't directly increase resources. Some of patterns (like map-reduce and pipe-filter) can better make use of increased resources than others.

Security

Pattern	Detect Attacks			Resist						React
	Detect Intrusion	Detect Service Denial	Verity Message Integrity	Identify	Authenticate	Authorize	Limit Access	Limit Exposure	Encryption	Revoke Access
Layered	X	X	X	X	X	X	X	X	X	X
Broker	X	X	X	X	X	X	X	X	X	X
MVC	X	X	X	X	X	X	X		X	
Pipe-filter			X	X	X	X	X	X	X	X
Client-server	X	X	X	X	X	X	X	X	X	X
P2P			X	X	X	X			X	
Service-oriented	X	X	X	X	X	X	X	X	X	X
Publish-subscribe	X	X	X	X	X	X	X	X	X	X
Shared-data	X	X	X	X	X	X	X	X	X	X
Multi-tier	X	X	X	X	X	X	X	X	X	X

Map-reduce			X						X	

Pattern	Benefits	Penalties
Layered	Can easily detect and react to attacks in each layer. Internal faults are not easily exposed.	More layers provide more potential faults to be abused.
Broker	The broker can monitor all traffic to the server, so it can detect, resist and react to attacks accordingly.	The broker itself might be attacked, which is hard to react to; the broker can be easily bypassed if some internal interfaces are compromised.
MVC	Each component can authenticate and authorize actors and act accordingly.	Since three components are all exposed, the attack surface can be big.
Pipe-filter	Each filter can check security problem and react. Usually the external interfaces are minimized, and the attack surface are minimized as well. Data can be encrypted.	Usually not all filters check security problem for sake of performance, so the filters that don't check can be the weak point of the system.
Client-server	The server should (and usually do) check security problem and react to it before it reaches later stages.	The server can be the easy entry point if its security strategy is bypassed or failed.
P2P	Some security strategy (like encryption and authentication) can be used to minimize the harm from attack.	A hostile peer can destroy the whole network; because of its wide-spread nature, most interfaces might be exposed; security fixes cannot be easily deployed
Service-oriented	Protocol and intermediary monitor traffic between two ends so two ends can detect and react to attacks effectively.	The middleware and protocol defects might be abused; the registry might be target which might cause a system failure.
Publish-subscribe	The event distributor can monitor and check security problems.	The compromise of the event distributor can be disastrous; a hostile event registrant can abuse the callback to issue an attack.
Shared-data	The data accessors and data store can deploy security strategies to react to attack.	Once the data store is compromised, all data accessors will be affected.
Multi-tier	Can detect and react to attacks in each tier to avoid attack from going deeper.	More tiers provide more potential faults to be abused.
Map-reduce	Can do some simple security checks.	Most map-reduce systems are optimized for performance so lacks enough security strategies and relies on other components for security.

Testability

Pattern	Control and Observe State			Limit Complexity	
	Specialized Interfaces	Record/Playback	Sandbox	Limit Structural Complexity	Limit Nondeterminism
Layered	X	X	X	X	X
Broker	X	X	X	X	X
MVC	X		X		
Pipe-filter	X	X	X	X	X
Client-server	X	X	X	X	X
P2P					
Service-oriented	X	X	X		X
Publish-	X	X	X		

subscribe					
Shared-data	X		X		
Multi-tier	X	X	X	X	X
Map-reduce	X	X	X	X	X

Pattern	Benefits	Penalties
Layered	Each layer has specialized responsibility and limited complexity and can be tested individually using mock and stubs.	Testing a layer may require great amount of codes for mock and stubs and sometimes unrealistic.
Broker	Server and client are independent with each other and can tested independently.	Broker involves nearly all interactions in the system and therefore is hard to test.
MVC	Each component has specialized responsibility and therefore can be tested.	Operation of each component relies on another two components and increases structural complexity and test difficulty.
Pipe-filter	Each filter focuses on one specific job and the dependency to environment is minimized, so can be easily tested.	The number of tests needed can be huge.
Client-server	The connection between server and client is only stable protocol so both can be individually tested.	The nondeterminism in production environment is hard to mimic during test so many prone defects may not be found.
P2P	P2P network is hard to test.	The environment in and nature of P2P network is so complex that it is nearly impossible to test.
Service-oriented	Each component has predefined and fixed responsibility and therefore can be tested individually.	It is hard to control independent services; the complicated structure leads to a complicated test suite to test each component.
Publish-subscribe	Each event can be tested in sandbox environment.	The event dispatching system may not be consistent in real production environment.
Shared-data	The logic of data accessors can be tested with mock data.	The shared data store increases nondeterminism and the degree of dependency to other components and is hard to predict and test.
Multi-tier	Each tier has specialized responsibility and limited complexity so can be tested like a layer.	Since a tier is usually much bigger and more complex than a layer, fully testing a tier requires even more codes and can be unrealistic.
Map-reduce	A map-reduce system is usually only for calculation so does not rely on external components too much, so can be tested with enough test data.	The data coming from production environment may not be as ideal and safe as data in testing is, so some potential faults might be ignored.

Usability

Pattern	Support User Initiative				Support System Initiative		
	Cancel	Undo	Pause/Resume	Aggregate	Maintain Task Model	Maintain User Model	Maintain System Model
Layered				X	X	X	X
Broker				X	X	X	X
MVC	X	X	X	X	X	X	

Pipe-filter	X		X	X	X	X	X
Client-server	X	X	X	X	X	X	X
P2P				X			
Service-oriented	X	X	X	X	X	X	X
Publish-subscribe				X			
Shared-data				X			
Multi-tier				X	X	X	X
Map-reduce				X	X	X	X

Pattern	Benefits	Penalties
Layered	System are separated into multiple specialized layers which each can predict user and system behavior according to their own job. (divide and conquer); implementation is hidden from user.	Providing user initiative (like undo functionality) may require coordination and interactions among multiple layers, adding to implementation difficulty.
Broker	A “almighty” broker can provide all services to user, making using service of the system much easier. The work is distributed by broker so that the server can determine context and behavior easier.	It is hard to determine and predict the behavior of a working broker.
MVC	User’s operation in view can be quickly processed by model and controller which improves user experiences.	All three parts in MVC contain states and behaviors on their own so it is hard to predict and determine their behavior.
Pipe-filter	Each pipe has specialized job and is easy to predict. User may monitor data at real time and can cancel or pause it as they wish.	It is hard to “undo” completed job; it has limited use cases: an interactive system may find a pipeline not suitable.
Client-server	A client-server structure is intuitive for user since most everyday work involves a similar stricture.	The server handling multiple concurrent requests might become unpredictable.
P2P	In some use cases, P2P is easy to use since user doesn’t have to care about how actually a complex request is completed in an equally complicated network.	A P2P network is unpredictable; user operation like canceling, undoing, pausing, and system prediction and state determination is nearly impossible to guarantee.
Service-oriented	Complicated abstraction ensures that each part of a complicated system is predictable and understandable. (divide and conquer)	Most individual might not be able to understand or use the whole system but a small part of it.
Publish-subscribe	Registering to an event and waiting for response makes using the system easier.	User’s operation may not always be satisfied and done in their expected way.
Shared-data	It is sometimes efficient to transfer data between user and the system.	Shared data increases unpredictability; user’s awareness of shared data store means user’s knowledge to the system is too much.
Multi-tier	Like layered, each tier of the system is predictable; implementation is hidden.	Like layered but worse, the cooperation between tiers might be costlier than layers.
Map-reduce	It significantly improves the ability to solve complicated problems for users; each map instance is stateless so it	The use cases are limited only on tasks with data that is of a large amount and can be divided.

* A good system design should hide implementation from being aware by end users; that is to say, user initiative should have nothing to do with the design pattern the system implementation actually applies.