The Reactive Manifesto

Glossary

Table of Contents

Asynchronous

The Oxford Dictionary defines asynchronous as “not existing or occurring at the same time”. In the context of this manifesto we mean that the processing of a request occurs at an arbitrary point in time, sometime after it has been transmitted from client to service. The client cannot directly observe, or synchronize with, the execution that occurs within the service. This is the antonym of synchronous processing which implies that the client only resumes its own execution once the service has processed the request.

Back-Pressure

When one component is struggling to keep-up, the system as a whole needs to respond in a sensible way. It is unacceptable for the component under stress to fail catastrophically or to drop messages in an uncontrolled fashion. Since it can’t cope and it can’t fail it should communicate the fact that it is under stress to upstream components and so get them to reduce the load. This back-pressure is an important feedback mechanism that allows systems to gracefully respond to load rather than collapse under it. The back-pressure may cascade all the way up to the user, at which point responsiveness may degrade, but this mechanism will ensure that the system is resilient under load, and will provide information that may allow the system itself to apply other resources to help distribute the load, see Elasticity.

Batching

Current computers are optimized for the repeated execution of the same task: instruction caches and branch prediction increase the number of instructions that can be processed per second while keeping the clock frequency unchanged. This means that giving different tasks to the same CPU core in rapid succession will not benefit from the full performance that could otherwise be achieved: if possible we should structure the program such that its execution alternates less frequently between different tasks. This can mean processing a set of data elements in batches, or it can mean performing different processing steps on dedicated hardware threads.

The same reasoning applies to the use of external resources that need synchronization and coordination. The I/O bandwidth offered by persistent storage devices can improve dramatically when issuing commands from a single thread (and thereby CPU core) instead of contending for bandwidth from all cores. Using a single entry point has the added advantage that operations can be reordered to better suit the optimal access patterns of the device (current storage devices perform better for linear than random access).

Additionally, batching provides the opportunity to share out the cost of expensive operations such as I/O or expensive computations. For example, packing multiple data items into the same network packet or disk block to increase efficiency and reduce utilisation.

Component

What we are describing is a modular software architecture, which is a very old idea, see for example Parnas (1972). We are using the term “component” due to its proximity with compartment, which implies that each component is self-contained, encapsulated and isolated from other components. This notion applies foremost to the runtime characteristics of the system, but it will typically also be reflected in the source code’s module structure as well. While different components might make use of the same software modules to perform common tasks, the program code that defines the top-level behavior of each component is then a module of its own. Component boundaries are often closely aligned with Bounded Contexts in the problem domain. This means that the system design tends to reflect the problem domain and so is easy to evolve, while retaining isolation. Message protocols provide a natural mapping and communications layer between Bounded Contexts (components).

Delegation

Delegating a task asynchronously to another component means that the execution of the task will take place in the context of that other component. This delegated context could entail running in a different error handling context, on a different thread, in a different process, or on a different network node, to name a few possibilities. The purpose of delegation is to hand over the processing responsibility of a task to another component so that the delegating component can perform other processing or optionally observe the progress of the delegated task in case additional action is required such as handling failure or reporting progress.

Elasticity (in contrast to Scalability)

Elasticity means that the throughput of a system scales up or down automatically to meet varying demand as resource is proportionally added or removed. The system needs to be scalable (see Scalability) to allow it to benefit from the dynamic addition, or removal, of resources at runtime. Elasticity therefore builds upon scalability and expands on it by adding the notion of automatic resource management.

Failure (in contrast to Error)

A failure is an unexpected event within a service that prevents it from continuing to function normally. A failure will generally prevent responses to the current, and possibly all following, client requests. This is in contrast with an error, which is an expected and coded-for condition—for example an error discovered during input validation, that will be communicated to the client as part of the normal processing of the message. Failures are unexpected and will require intervention before the system can resume at the same level of operation. This does not mean that failures are always fatal, rather that some capacity of the system will be reduced following a failure. Errors are an expected part of normal operations, are dealt with immediately and the system will continue to operate at the same capacity following an error.

Examples of failures are hardware malfunction, processes terminating due to fatal resource exhaustion, program defects that result in corrupted internal state.

Isolation (and Containment)

Isolation can be defined in terms of decoupling, both in time and space. Decoupling in time means that the sender and receiver can have independent life-cycles—they do not need to be present at the same time for communication to be possible. It is enabled by adding asynchronous boundaries between the components, communicating through message-passing. Decoupling in space (defined as Location Transparency) means that the sender and receiver do not have to run in the same process, but wherever the operations division or the runtime itself decides is most efficient—which might change during an application's lifetime.

True isolation goes beyond the notion of encapsulation found in most object-oriented languages and gives us compartmentalization and containment of:

- State and behavior: it enables share-nothing designs and minimizes contention and coherence cost (as defined in the Universal Scalability Law);
- Failures: it allows failures to be captured, signalled and managed at a fine-grained level instead of letting them cascade to other components.

Strong isolation between components is built on communication over well-defined protocols and enables loose coupling, leading to systems that are easier to understand, extend, test and evolve.

Location Transparency

Elastic systems need to be adaptive and continuously react to changes in demand, they need to gracefully and efficiently increase and decrease scale. One key insight that simplifies this problem immensely is to realize that we are all doing distributed computing. This is true whether we are running our systems on a single node (with multiple independent CPUs communicating over the QPI link) or on a cluster of nodes (with independent machines communicating over the network). Embracing this fact means that there is no conceptual difference between scaling vertically on multicore or horizontally on the cluster.

If all of our components support mobility, and local communication is just an optimization, then we do not have to define a static system topology and deployment model upfront. We can leave this decision to the operations personnel and the runtime, which can adapt and optimize the system depending on how it is used.

This decoupling in space (see the the definition for Isolation), enabled through asynchronous message-passing, and decoupling of the runtime instances from their references is what we call Location Transparency. Location Transparency is often mistaken for 'transparent distributed computing', while it is actually the opposite: we embrace the network and all its constraints—like partial failure, network splits, dropped messages, and its asynchronous and message-based nature—by making them first class in the programming model, instead of trying to emulate in-process method dispatch on the network (ala RPC, XA etc.). Our view of Location Transparency is in perfect agreement with A Note On Distributed Computing by Waldo et al.

Message-Driven (in contrast to Event-Driven)

A message is an item of data that is sent to a specific destination. An event is a signal emitted by a component upon reaching a given state. In a message-driven system addressable recipients await the arrival of messages and react to them, otherwise lying dormant. In an event-driven system notification listeners are attached to the sources of events such that they are invoked when the event is emitted. This means that an event-driven system focuses on addressable event sources while a message-driven system concentrates on addressable recipients. A message can contain an encoded event as its payload.

Resilience is more difficult to achieve in an event-driven system due to the short-lived nature of event consumption chains: when processing is set in motion and listeners are attached in order to react to and transform the result, these listeners typically handle success or failure directly and in the sense of reporting back to the original client. Responding to the failure of a component in order to restore its proper function, on the other hand, requires a treatment of these failures that is not tied to ephemeral client requests, but that responds to the overall component health state.

Non-Blocking

In concurrent programming an algorithm is considered non-blocking if threads competing for a resource do not have their execution indefinitely postponed by mutual exclusion protecting that resource. In practice this usually manifests as an API that allows access to the resource if it is available otherwise it immediately returns informing the caller that the resource is not currently available or that the operation has been initiated and not yet completed. A non-blocking API to a resource allows the caller the option to do other work rather than be blocked waiting on the resource to become available. This may be complemented by allowing the client of the resource to register for getting notified when the resource is available or the operation has completed.

Protocol

A protocol defines the treatment and etiquette for the exchange or transmission of messages between components. Protocols are formulated as relations between the participants to the exchange, the accumulated state of the protocol and the allowed set of messages to be sent. This means that a protocol describes which messages a participant may send to another participant at any given point in time. Protocols can be classified by the shape of the exchange, some common classes are request–reply, repeated request–reply (as in HTTP), publish–subscribe, and stream (both push and pull).

In comparison to local programming interfaces a protocol is more generic since it can include more than two participants and it foresees a progression of the state of the message exchange; an interface only specifies one interaction at a time between the caller and the receiver.

It should be noted that a protocol as defined here just specifies which messages may be sent, but not how they are sent: encoding, decoding (i.e. codecs), and transport mechanisms are implementation details that are transparent to the components' use of the protocol.

Replication

Executing a component simultaneously in different places is referred to as replication. This can mean executing on different threads or thread pools, processes, network nodes, or computing centers. Replication offers scalability, where the incoming workload is distributed across multiple instances of a component, or resilience, where the incoming workload is replicated to multiple instances which process the same requests in parallel. These approaches can be mixed, for example by ensuring that all transactions pertaining to a certain user of the component will be executed by two instances while the total number of instances varies with the incoming load, (see Elasticity).

Resource

Everything that a component relies upon to perform its function is a resource that must be provisioned according to the component's needs. This includes CPU allocation, main memory and persistent storage as well as network bandwidth, main memory bandwidth, CPU caches, inter-socket CPU links, reliable timer and task scheduling services, other input and output devices, external services like databases or network file systems etc. The elasticity and resilience of all these resources must be considered, since the lack of a required resource will prevent the component from functioning when required.

Scalability

The ability of a system to make use of more computing resources in order to increase its performance is measured by the ratio of throughput gain to resource increase. A perfectly scalable system is characterized by both numbers being proportional: a twofold allocation of resources will double the throughput. Scalability is typically limited by the introduction of bottlenecks or synchronization points within the system, leading to constrained scalability, see Amdahl's Law and Gunther's Universal Scalability Model.

System

A system provides services to its users or clients. Systems can be large or small, in which case they comprise many or just a few components. All components of a system collaborate to provide these services. In many cases the components are in a client–server relationship within the same system (consider for example front-end components relying upon back-end components). A system shares a common resilience model, by which we mean that failure of a component is handled within the system, delegated from one component to the other. It is useful to view groups of components within a system as subsystems if they are isolated from the rest of the system in their function, resources or failure modes.

User

We use this term informally to refer to any consumer of a service, be that a human or another service.

↑ Back To Top