Università della Svizzera italiana | Faculty of Informatics

Architectural Patterns (3)

Prof. Cesare Pautasso
http://www.pautasso.info
cesare.pautasso@usi.ch
@pautasso

Scatter/Gather

Goal: send the same message to multiple recipients which will (or may) reply to it

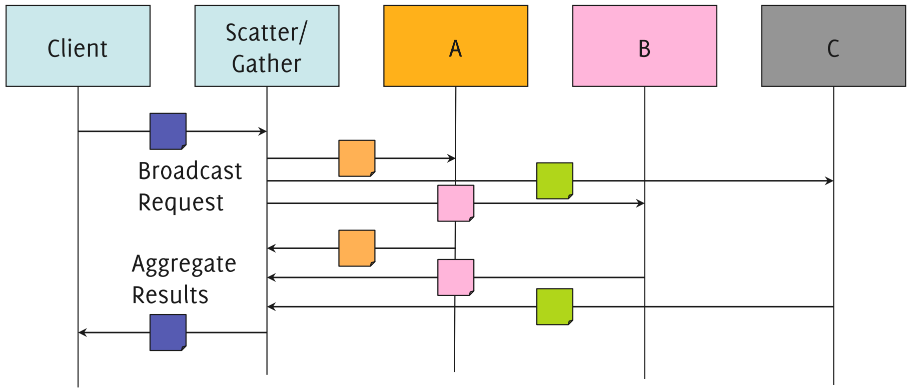
combine the notification of the request with aggregation of replies

Pattern: broadcast a message to all recipients, wait for all (or some) answers and aggregate them into a single message

Composition

1. Scatter/Gather
2. Canary Call
3. Master/Slave
4. Load Balancing
5. Composition

Scatter/Gather



Broadcasting can be implemented using subscriptions (loosely coupled) or a distribution list provided by the client (which knows A, B, C)
Results will be collected and aggregated before they are returned to the original client as a single message

Scatter/Gather

- This is a simple composition pattern, where we are interested in aggregating replies of the components processing the same message.
- Alternatives for controlling the components:
  - The recipients can be discovered (using subscriptions) or be known a priori (distribution list attached to request)
- Warning: the response-time of each component may vary and the response-time of the scatter/gather is the slowest of all component responses
- Different synchronization strategies:
  - Wait for all messages
  - Wait for some messages (within a certain time window, or using an N-out-of-M synchronization)
  - Return fastest reply (discriminator, maybe only under certain conditions)
- Example:
  - Contact N airlines simultaneously for price quotes
  - Buy ticket from either airline if price<=200 CHF
  - Buy the cheapest ticket if price >200 CHF
  - Make the decision within 2 minutes

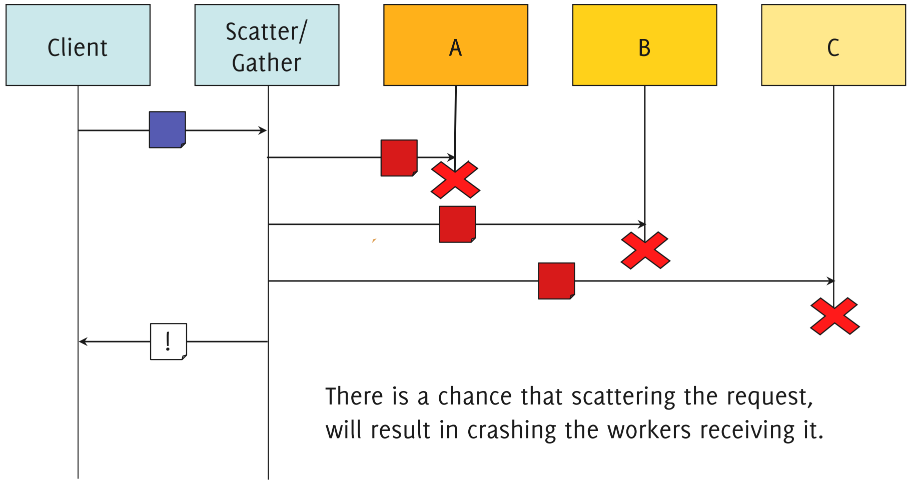
Canary Call

Goal: avoid crashing all recipients of a poisoned request

use an heuristic to evaluate the request

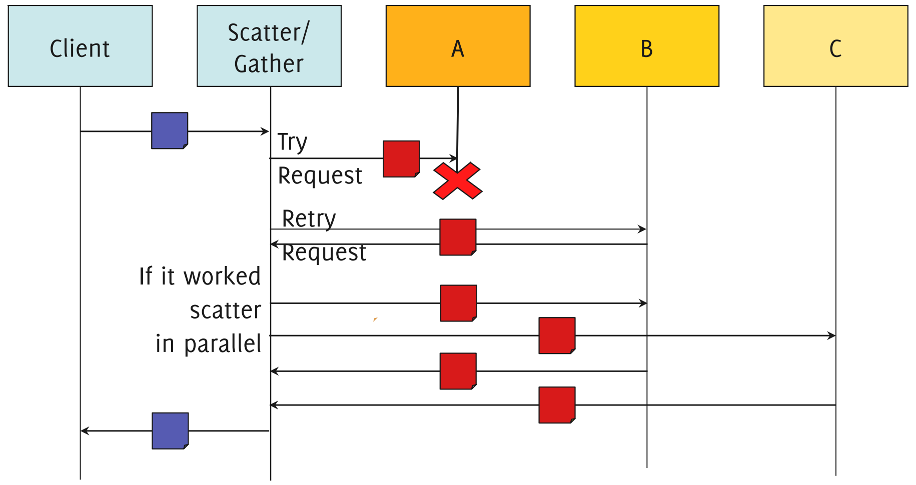
Pattern: try the potentially dangerous requests on one recipient and scatter the request only if it survives

Canary Call



There is a chance that scattering the request, will result in crashing the workers receiving it.

Canary Call



The first parallel request is sent before all others to check if it would harm the worker receiving it (canary request).
The decision on whether all other requests should be scattered depends on an heuristics: continue if the canary request is successful after some attempts.

- Performance/Robustness Trade-Off:
  - Decreased Performance: the scatter phase waits for the canary call to succeed (response time doubles)
  - Increase Robustness: most workers survive a poisonous call, which would have failed anyway
- Apply when worker recovery is expensive, or when there are thousands of workers involved
- Heuristic: Failed canary calls are not necessarily poisonous

Master/Slave

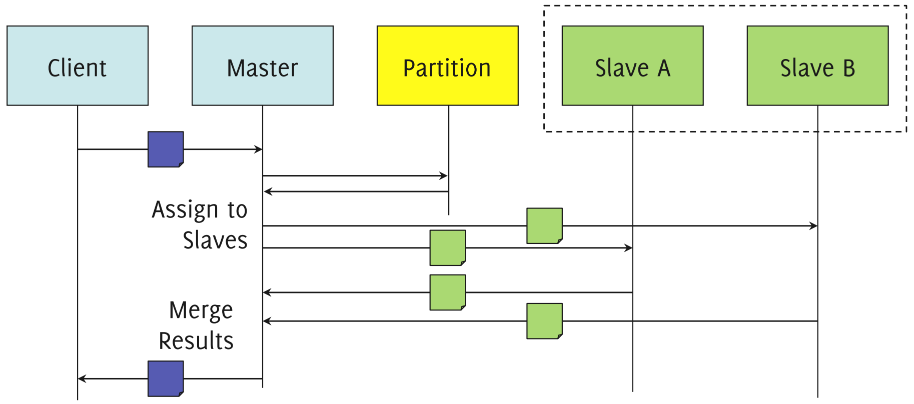
Goal: speed up the execution of long running computations

split a large job into smaller independent partitions which can be processed in parallel

Pattern: the master divides the work among a pool of slaves and gathers the results once they arrive.

Synonyms: Master/Worker, Divide-and-Conquer.

Master/Slave



Assignment to slaves can be implemented using push or pull.
The set of slaves is not known in advance and may change over time.
Results will be collected and aggregated before they are returned to the original client

Master/Slave

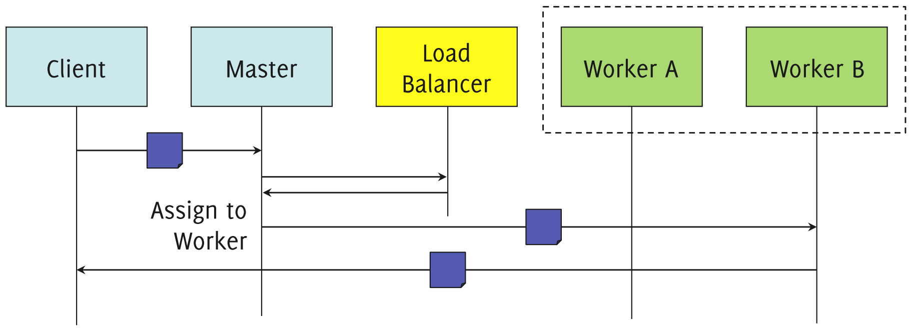
- This composition pattern is a specialized version of the scatter/gather pattern.
- Clients should not know that the master delegates its task to a set of slaves
- Master Properties:
  - Different partitioning strategies may be used
  - Fault Tolerance: if a slave fails, resend its partition to another one
  - Computational Accuracy: send the same partition to multiple slaves and compare their results to detect inaccuracies (this works only if slaves are deterministic)
  - Master is application independent
- Slave Properties:
  - Each slave runs its own thread of control
  - Slaves may be distributed across the network
  - Slaves may join and leave the system at any time (may even fail)
  - Slaves do not usually exchange any information among themselves
  - Slaves should be independent from the algorithm used to partition the work
- Example Applications:
  - Matrix Multiplication (compute each row independently)
  - Movie Rendering (compute each picture frame independently)
  - TSP Combinatorial Optimization (local vs. global search)

Load Balancing

Goal: speed up and scale up the execution of multiple requests of many clients

deploy many replicated instances of the server on multiple machines

Pattern: the master assigns the requests among a pool of workers, which answer directly to the clients.



Different load balancing policies can be used by the master to distribute requests among a pool of workers.
The set of workers is not known in advance and may change over time.
Results will be sent directly by the worker back to the client
(but may have to go through the master)

Load Balancing

- This composition pattern is similar to the master/worker pattern, but there is no partitioning of the request nor aggregation of the response
- Clients should not know that the master delegates its task to a set of workers
- Master Properties:
  - Different load balancing policies may be used
  - Fault Tolerance: if a slave fails, resend the request to another one (assuming idempotent failed requests can be detected by the master)
  - Master and load balancing policies are application independent
- Worker Properties:
  - Each worker runs its own thread of control
  - Workers may be distributed across the network
  - Workers may join and leave the system at any time (may even fail)
  - Workers do not usually exchange any information among themselves
  - Workers should be independent from the algorithm used to load balance the requests
- Variants:
  - Stateless load-balancing (every request from any client goes to any worker)
  - Session-based load-balancing (requests from the same client always go to the same worker)
  - Elastic load balancing (the pool of workers is dynamically resized based on the amount of work)

Composition

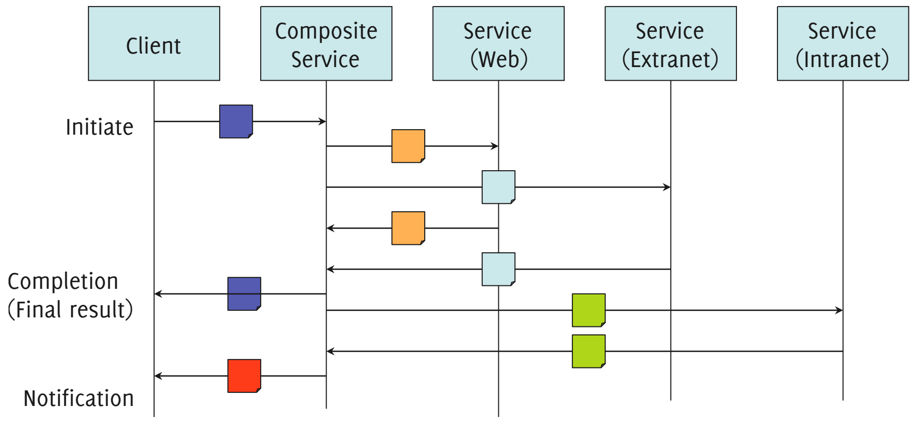
Goal: Improve reuse of existing applications

Build systems out of the composition of existing ones

Pattern: by including compositions as an explicit part of the architecture, it becomes possible to reuse existing services in order to aggregate basic services into value-added ones

Variants: Synchronous, Asynchronous

Synonym: Orchestration



Composite services provide value-added services to clients by aggregating a set of services and orchestrating them according to a well-defined and explicit business process model

Composition

- Composition is recursive: a composite service is a service that can be composed.
- Services involved in a composition do not necessarily have to know that they are being orchestrated as part of another service and may be provided by different organizations
- Services can be reused across different compositions and compositions can be reused by binding them to different services
- Composite services may be implemented using a variety of tools and techniques:
  - Ordinary programming languages (Java, C#, ...)
  - Scripting languages (Python, PERL, ...)
  - Workflow modeling languages (JOpera, BPEL...)
  - Compositions may be long-running processes and involve asynchronous interactions (both on the client side, and with the services)

Free Advice

The best architectures are full of patterns

Do not use too many unnecessary patterns
(this is an anti-pattern)

More Free Advice

Most software systems cannot be structured according to a single architectural pattern.

Choosing one or more patterns does **not** give a complete software architecture.
Further refinement is needed.

References

- Paul Monday, Web Service Patterns: Java Edition, APress, 2003
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison- Wesley, 1994
- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Patterns-Oriented Software Architecture, Vol. 1: A System of Patterns, John Wiley, 1996
- Martin Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2003
- Gregor Hohpe, Bobby Woolf, Enterprise Integration Patterns, Addison Wesley, 2004