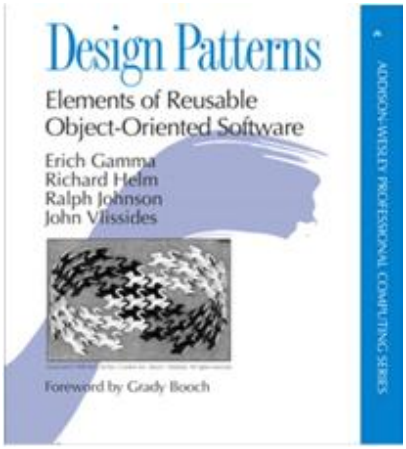
7 Design Patterns for Almost-infinite Scalability

Thursday, December 16, 2010 at 8:02AM

General Chicken in Strategy



Good article from manageability.com summarizing design patterns from Pat Helland's amazing paper Life beyond Distributed Transactions: an Apostate's Opinion.

1. **Entities are uniquely identified** - each entity which represents disjoint data (i.e. no overlap of data between entities) should have a unique key.
2. **Multiple disjoint scopes of transactional serializability** - in other words there are these 'entities' and that you cannot perform atomic transactions across these entities.
3. **At-Least-Once messaging** - that is an application must tolerate message retries and out-of-order arrival of messages.
4. **Messages are adressed to entities** - that is one can't abstract away from the business logic the existence of the unique keys for addresing entities. Addressing however is independent of location.
5. **Entities manage conversational state per party** - that is, to ensure idemptency an entity needs to remember that a message has been previously processed. Furthermore, in a world without atomic transactions, outcomes need to be 'negotiated' using some kind of workflow capability.
6. **Alternate indexes cannot reside within a single scope of serializability** - that is, one can't assume the indices or references to entities can be update atomically. There is the potential that these indices may become out of sync.

7. **Messaging between Entities are Tentative** - that is, entities need to accept some level of uncertainty and that messages that are sent are requests form commitment and may possibly be cancelled.

The article then compares how these principles compare to the design principles used to develop S3:

- **Decentralization:** Use fully decentralized techniques to remove scaling bottlenecks and single points of failure.
- **Asynchrony:** The system makes progress under all circumstances.
- **Autonomy:** The system is designed such that individual components can make decisions based on local information.
- **Local responsibility:** Each individual component is responsible for achieving its consistency; this is never the burden of its peers.
- **Controlled concurrency:** Operations are designed such that no or limited concurrency control is required.
- **Failure tolerant:** The system considers the failure of components to be a normal mode of operation, and continues operation with no or minimal interruption.
- **Controlled parallelism:** Abstractions used in the system are of such granularity that parallelism can be used to improve performance and robustness of recovery or the introduction of new nodes.
- **Decompose into small well-understood building blocks:** Do not try to provide a single service that does everything for every one, but instead build small components that can be used as building blocks for other services.
- **Symmetry:** Nodes in the system are identical in terms of functionality, and require no or minimal node-specific configuration to function.
- **Simplicity:** The system should be made as simple as possible (- but no simpler).
