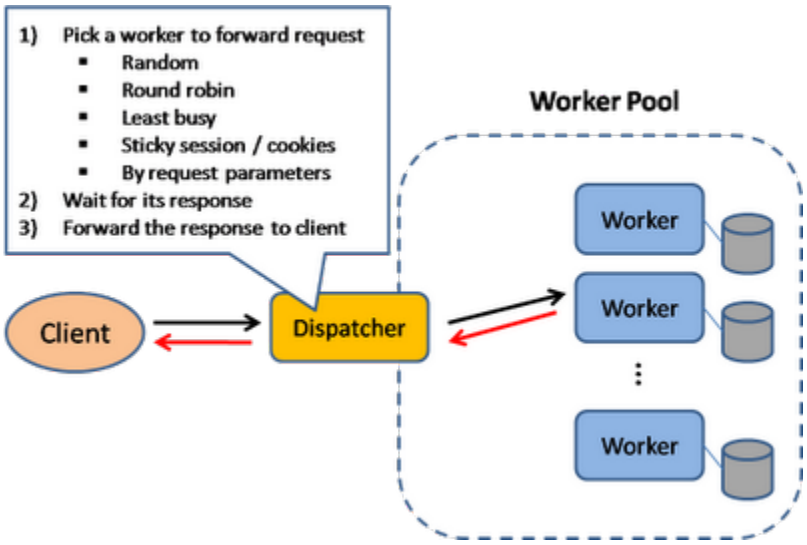
Patterns in network system design （续）

前一段时间写了一系列"Patterns in network system design"的文章，最近又继续思考这个话题，感觉还有一些 Pattern 遗漏了。Pattern 的发现，整理，是一个持续的过程。随着实践的深入和系统的发展，应该会有更多的 pattern 被发现，而以前的 pattern 有可能在新系统里面失去了作用。所以要持续不断的优化，整理自己的知识，使之与现实的系统能够匹配。

提出（发现）问题；分析问题；解决问题是一个完整的过程。很多时候，解决问题并不困难，难的是在分析问题和发现问题。对我来说，提出（发现）问题，尤其重要，这代表了不断思考，不断拓展知识的兴趣。只有对一件事情有兴趣，才能对分析和解决问题有持续的热情，这才是一个良性的循环。

35) Load balance

Pattern Name and Classification: load balance
Intent: 把任务均衡分配给 worker
Also Known As: No
Motivation (Forces):



(图来自 http://www.kernelchina.org/?q=node/552，借来用一下，这个图是在描述 distributed system 里面的 load balance，但是也适用于网络系统，不同的系统，解决问题的思路有时是一样的）如果系统里面有多个计算单元，如何把任务分配给每个计算单元，使得每个计算单元的负载是平衡的，这就是负载均衡要考虑的问题，有很多不同的调度策略（这里要考虑机制（mechanism）和策略（policy）分离，策略是可以替换的，但最好是一致的，换句话说，就是相同的策略才能产生一致的结果，否则就会引入新的不均衡）。如果计算单元的计算能力是一样的，round-robin 应该是首选；如果是不一样的，可以选择 weighted round-robin。但是，有些时候，把 session 分配到哪个计算单元上是由应用决定的（相当于另一种策略），比如 ALG 的 control session 需要和 data session 在一起；又比如说 http transaction stickness（一个 transaction 需要在同一个计算单元上，很多应用有这样的要求）。这样会导致系统的负载难以预料。这时，需要在系统里面加一些探测机制（系统是否可用）和反馈机制（计算单元的负载情况），并根据这些结果调整负载的分配。由于没有一个适用于所有情况的调度策略，所以做 load balance 很难。load balance 不保证 100%正确的，比如把 packet 分配到某个计算单元，由于计算单元的负载太大而不能处理而导致的丢包，这

种情况是很难避免，而且这个 packet 也不能送回去重新调度，否则会带来更多的错误。
Known Uses: load balancer；packet scheduler
Related Patterns: centralized design；decentralized design
References:

1: http://zh.linuxvirtualserver.org/

2: http://www.f5.com.cn/

3: http://www.a10networks.com/

36) Ordering

Pattern Name and Classification: ordering
Intent: 保证 packet 的进出顺序
Also Known As: No
Motivation (Forces):

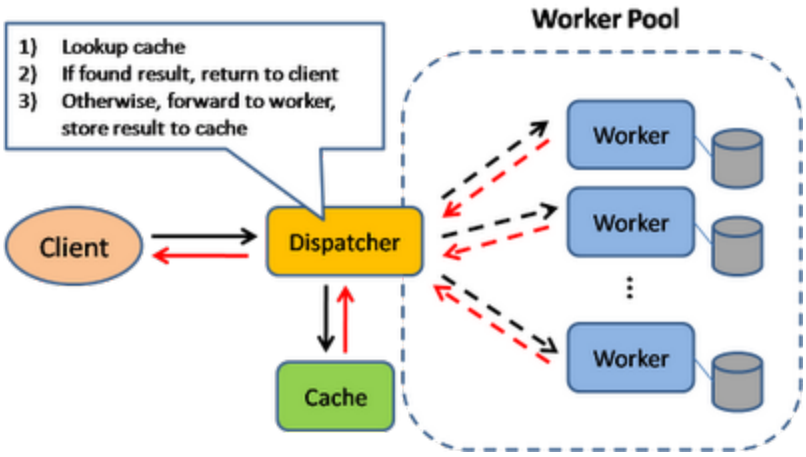
首先要问题两个问题：1）为什么要保序？2）保什么样的序？

按道理说，IP packet 本身并不要求按顺序处理（虽然有 ipid，但除了 fragment 会用到外，别的并不关心，而且在 ipv6 里面，已经把 ipid 从 ip 头里面去掉了），为什么还需要保序哪（保序是针对网关而言）？保序是上层应用的要求，不管是 TCP 还是 UDP。有很多 UDP 应用，需要按顺序处理 packet，否则就会出错；而对 TCP 来说，顺序对 TCP 的性能有很大影响。对网关来说，最低的要求是不能引入新的乱序，也就是说，packet 进出网关的顺序是一致的。当然，packet 的顺序对一个 stream 是有效的，并不要求 packet 的全局顺序。只要保证一个 stream 的 packet 进出网关的顺序是一致的，就基本满足了要求。ordering 在单处理器上是没有问题的；但是在多核和多机框的情况下，就会引入乱序的问题。所有很多多核网络处理器都会有保序的硬件单元，当然也可以软件实现。保序是一个串行的动作，会降低系统的处理速度。
Known Uses: multicore; multi-chassis system
Related Patterns: No
References:

1: http://en.wikipedia.org/wiki/Out-of-order_delivery

37) Cache

Pattern Name and Classification: cache
Intent: 优化系统性能
Also Known As: No
Motivation (Forces):



（图来自 http://www.kernelchina.org/?q=node/552）

Cache 做为一种优化性能的手段，在很多地方都会用到。不管是 CPU cache，还是 http proxy，亦或者是 DNS cache 等等。为什么 cache 能够提高性能，主要原因就是局部性（locality）：包括时间局部性（temporal locality）和空间局部性（spatial locality）。Cache 所面对的问题也是一样的，包括：

- Cache size：每个 cache entry 的大小，有多少个 cache entry，cache associate（关联）是多少，cache 并不是越大越好，cache 容量增大，会导致管理复杂，性能会下降

- Cache miss：增加 cache hit，降低 cache miss 是设计 cache 的目标
- Cache conflict：cache 容量有限，所以 cache 会冲突，在冲突的时候，需要替换 cache，就会有不同的替换策略，cache 同样也有生命期，cache 并不是一直有效的，比如 http cache, dns cache 就有一个有效期

Known Uses: cpu cache, http cache, dns cache
Related Patterns: No
References:

1：http://en.wikipedia.org/wiki/Cache

2：http://en.wikipedia.org/wiki/DNS_cache

3：http://en.wikipedia.org/wiki/Content_delivery_network

38) Pre allocation

Pattern Name and Classification: pre allocation
Intent: 预分配内存
Also Known As: memory reservation
Motivation (Forces):

Data plane 的内存管理需要简单，所以很多时候都采用预分配的策略。每个模块在初始化是预分配自己的内存，然后初始化成 resource pool。这样简化了内存管理的工作，可以提高性能。由于 resource pool 里面的 resource 的数据结构是一样的，这样对 cache 也有一定的好处。

预分配内存可以保证系统的容量（capacity）。但是坏处也很明显，就是被预分配的内存即使没有被用到，也没办法释放给别人用，这样会浪费内存。特别是在系统有很多模块的时候。但是，如果采用动态分配内存方式，系统性能就会受影响，如何在静态和动态之间达到平衡，是需要考虑的问题。静态加动态，应该是解决这个问题的一个思路，不过需要试一试才知道。
Known Uses: data plane memory management
Related Patterns: No
References:

1: http://en.wikipedia.org/wiki/Memory_management

2: http://linux-mm.org/

39) Classify

Pattern Name and Classification: classify
Intent: 分类处理
Also Known As: No
Motivation (Forces):

一般情况下，网络设备的流程都会有 parser, classify, process。分类（classify）是很重要的一个环节。为什么需要分类哪？这个和网络协议有一定的关系，比如 de-multiplex 就是一个分类的过程；而 multiplex 则与之相反，是个集中的过程。当把一个 packet 分到某一类之后，在 packet 需要做的动作也就明确了。把分类和处理分离开，可以简化设计，提高系统的可读性。

分类和模式识别，或者 DPI 是不是类似。个人认为是相同的过程。它基本上就是把未知的 packet，划分到已知的类别里面，然后再根据类别做相应的处理。已知的类别，需要一些特征来描述；而分类的过程就是用 packet 里面的一些数据和这些特征来比较。
Known Uses: 网络设备
Related Patterns: stateful processing
References:

1：http://en.wikipedia.org/wiki/Biological_classification

2：http://en.wikipedia.org/wiki/Pattern_matching

3：http://en.wikipedia.org/wiki/Deep_packet_inspection

40) State machine

Pattern Name and Classification: state machine
Intent: 运用状态机简化设计
Also Known As: No
Motivation (Forces):

状态机在协议分析和处理里面有很多应用，最常见是 TCP 的状态机。状态机由两个元素组成：状态；以及状态迁移。状态迁移是由动作引起的，因此一个状态机可以表示为 state machine = {event, action}。只要画出一个二维表，就能分析系统所有可能的路径，而且不会有遗漏。状态机是一个很好的工具，要学会运用这个工具解决实际的问题。
Known Uses: tcp
Related Patterns: No
References:

1: http://en.wikipedia.org/wiki/State_machine

2: http://en.wikipedia.org/wiki/File:Tcp_state_diagram_fixed_new.svg

41) Push and pull

Pattern Name and Classification: push and pull
Intent: 发送消息，更新状态
Also Known As: No
Motivation (Forces):

推（push）和拉（pull）是两个动作。push 是状态变化时，主动通知对方，当然前提是对方对自己的状态感兴趣（所以应该有一个注册过程）；而 pull 是主动查询对方的状态，可以定时查询，也可以在收到某个消息时查询。在设计模式里面有个 Observer 模式，和这个比较类似，在 observer 里面，用的是 push 的方式。在 load balance 里面监控计算单元的状态时，可以用 pull 的方式。
Known Uses: health monitor
Related Patterns: load balance；state machine；Observer
References:

1：http://en.wikipedia.org/wiki/State_observer

2：http://en.wikipedia.org/wiki/Observer_pattern

系统的性能，个人认为有三个层次的决定因素：

a) 系统层次：系统层次的设计，决定的系统的性能。什么是系统层次的设计？比如 core 的分配，数据的流程，比如软硬件的分配等等。在大的系统结构图画出来之后，系统的最佳性能应该就决定了，这是最难的部分。

b) 算法层次：算法的选择和优化，决定了系统能否达到最优性能，一般来说，系统只能接近最优性能，因为 O(1)的算法很少，特别在网络系统里面，很少有 O(1)的算法可能选择。需要选择平均性能好的算法，而不是最优和最差相差很大的算法。如果只是某些情况下算法性能好，那么最好不要在通用系统里面用它。

c) 代码层次：代码层次有很多优化技巧，比如 cache 优化，这方面的优化更多的是需要实验，因为某个优化可能只对某些测试 用例有效，而不是对所有测试用例有效。