读 CRAQ 论文的原因有两个：

1. 其为了能够容错进行了复制
2. 其链式复制机制与我们之前看到的（Raft）不一样

CRAQ 通过将读请求分给任意 replica 来提高读请求的吞吐量，读性能会随着 replica 数量的增加而增加。CRAQ 与 Zookeeper 不同，Zookeeper 为了能够从任意 replica 中读取数据，它们就得牺牲数据的即时性，因此，Zookeeper 就不具备线性一致性。但 CRAQ 也是从任意 replica 上读取数据，但它保留了强一致性

链式复制其实是一种方案。假设有多个副本，你想确保它们收到的都是相同顺序的写操作。假设有一连串服务器，这条链上的第一个服务器称为 head，最后一个则叫 tail。**当一个 client 想进行写操作的时候，它总是将所有的写请求发送给 head**，head 会将该当前数据更新或替换为 client 所写入数据的当前副本。然后该写请求沿着该链进行传播，当每个节点看到这个写请求的时候，它就会将新的数据写入该节点的数据副本中，**当写请求到达 tail 时，tail 会对 client 进行响应，并说，我们已经处理完了你的写请求**，这就是 CRAQ 处理写请求的方式

下面看看读请求，如果 client 想进行一次读请求，它会向 tail 发送读请求，tail 会将它当前版本的数据作为响应发送给 client，假如想读取 B，则 tail 就会将当前值 B 返回给我们

上面介绍的并不是 CRAQ，而是链式复制（chain replication）

在没有故障的情况下，服务器看到了所有写请求，也看到了所有读请求，一个读请求会看到刚被写入的最新的值，因此具有一致性

下面来看看出现故障的情况，发生故障后你看到的状态相对来说会受到限制。为了将写请求传给 tail，该写请求必须被这条链中的所有节点处理，所以要想该结果暴露出来用于 read，直到 tail 处上整个路径的节点都必须知道这个写请求

如果某个写请求没有被提交的话，这意味着在发生任何会扰乱系统的崩溃前，该写请求已经传递到崩溃点之前的所有节点，并没有传递到崩溃点之后的节点

- 如果是 head 节点故障，那么下一个节点可以简单地接手 head 的工作就行了，因为当 head 发生了故障，那么任何 client 发起的写请求就会发给第二个节点，然后写请求继续往下传递，最终我们会提交该请求，因为即便崩溃前 head 收到了该写请求，由于未转发给其他节点，因此该写请求肯定未提交，客户端不会收到确认信息
- 如果是 tail 节点故障，那么 tail 的前一个节点接手 tail 的工作
- 如果是 head 和 tail 之间的节点发生了故障，我们会将这个故障节点移除，然后前一个节点可能得重新发送最近的写请求给它的下一个新节点

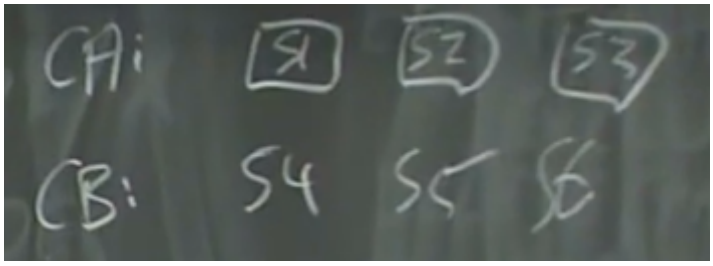
总而言之，这就是故障恢复

类似于 Raft （即 leader 向所有 client 发送信息）的代价会非常昂贵，因此 Raft 的 leader 的工作负载要比 chain replication leader 高得多，随着 client 请求数量的增加，raft leader 会慢慢达到处理能力的瓶颈，并且它的处理速度不再比 chain replication head 来的更快

另一个不同之处在于，在 raft 中，所有的读请求都需要由 leader 来进行处理，leader 会看到每个来自 client 端的请求。而链式复制中只有 tail 才会看到读请求

链式复制也有避免重复执行请求的机制，即对每个标号进行标识，如果 tail 节点挂了，前一个节点接收 tail 的工作，此时写请求刚好没有到达 tail 节点，客户端收不到响应，因此客户端可能会重发请求，然后 head 发现是已经执行过的请求，再进行回应

下面来看一个问题：即如果 head 不能和下一个节点通信，这样 head 会觉得剩下的节点挂掉了，但下一个节点是觉得 head 挂掉了，此时就会发生脑裂情况。其实现实中我们不能单独使用这种链式复制，我们还需要第三方的一些东西来判断哪些服务器还活着或者挂掉了。以此来让大家不再对链上的节点有分歧。这通常叫做配置管理器，负责去监控节点是否存活。每当配置管理器觉得某个服务器挂掉了，他就会对外发送一个新的配置，并且重新定义 head 和 tail。通常你的配置管理器是基于 raft、paxos 之类的构建的，其具备容错能力，并且不会收到脑裂影响



配置管理器会去决定这条链看起来应该是怎么样的，比如说 Chain A、Chain B 分别由三台不同的服务器组成，配置管理器会将这个组成告诉所有人，这样 client 知道了，其他服务器也知道了，这样服务器就不需要去关心其他服务器是否挂了，其只需要不断地和下一个服务器通信，即便挂了也是一直通信，直到收到配置服务器的新配置提醒为止

工业应用可能是对数据进行 hash 或者别的方式进行分区，然后到不同的链上去执行。

我们倾向于使用 raft 或 paxos 的一个原因是，它们无需去等待一个落后的 replica，链式复制中如果有一个 replica 速度很慢，那么它就会降低所有写操作处理的进度

只需要 majority 的要求可以一定程度提升我们的性能

如果配置管理器认为 head 和下一个节点都还活着，但是这两个节点都无法通信怎么办？**这是一个大问题**，我们需要对配置管理器做一些更严谨的配置，即确保这些节点能与配置服务器通信的同时，这些节点之间也能互相通信

Q: The paper's Introduction mentions that one could use multiple chains to solve the problem of intermediate chain nodes not serving reads. What does this mean?

A: In Chain Replication, only the head and tail directly serve client requests; the other replicas help fault tolerance but not performance. Since the load on the head and tail is thus likely to be higher than the load on intermediate nodes, you could get into a situation where performance is bottlenecked by head/tail, yet there is plenty of idle CPU available in the intermediate nodes. CRAQ exploits that idle CPU by moving the read work to them.

The Introduction is referring to this alternate approach. A data center will probably have lots of distinct CR chains, each serving a fraction (shard) of the objects. Suppose you have three servers (S1, S2, and S3) and three chains (C1, C2, C3). Then you can have the three chains be:

C1: S1 S2 S3
C2: S2 S3 S1
C3: S3 S1 S2

Now, assuming activity on the three chains is roughly equal, the load on the three servers will also be roughly equal. In particular the load of serving client requests (head and tail) will be roughly equally divided among the three servers.

This is a pretty reasonable arrangement; CRAQ is only better if it turns out that some chains see more load than others.

Q: Why does CRAQ keep the old clean object when it sees a write and creates a new dirty object?

A: Suppose a node has clean version 1, and dirty version 2. If the node receives a read from a client, it sends a "version query" to the tail. If the tail hasn't seen version 2, the tail will reply with version number 1. The node should then reply with the data for version 1. So it has to hold on to a copy of version 1's data until version 2 commits.