etcd vs Zookeeper

In general, assume that "reliable" clients to any consensus system need to follow a well thought out recipe to handle all errors gracefully. It is also safe to assume that the number of "reliable" clients is probably going to be quite small. Regardless of how the system is implemented, treating it as a reliable, highly available network file system for small objects is likely to result in fewer problems.

zkocc is a Zookeeper Open Connection Cache - it's a smart-ish caching proxy for Zookeeper that is part of vitess (github.com/youtube/vitess).

Zookeeper Pros

- Mature -- issues are a known quantity and most big bugs can be worked around in client code
- Well considered API and straightforward for the most part
- Software is generally reliable and uptime is largely determined by box or users abuse
- Many handy bits of functionality (sequences, ephemeral nodes)
- Known to handle large number of connections (20-30k per process) relatively well, even in a transcontinental setup

Zookeeper Cons

- Connections are heavyweight and expensive to construct. They require a quorum disk write on every connect/disconnect.
    - Solved by zkocc.
- Protocol is very complex, even when you just want to read/write a value.
    - Solved by zkocc.
- AUTH/ACLs are very poorly documented and rely on plain-text AUTH. However, they are enough to prevent accidental stomping of multiple apps or multiple dopey users.
- Complex to secure - you need to use IPsec, or stunnel.
- Quorum required for even for reads
    - Solve by zkocc which allows dirty reads for keys you have already requested.
- Large data recovery can be difficult. Timeouts for internal leader election don't automatically scale to take into account the size of the recovery image, leading to outages that are hard to recover from without lots of manual intervention. (initLimit).
- Terrible logging - poor classification of warning vs information (info is too verbose, warning gives you no insight)
- Too many timeouts and poor documentation about how they interact and (fail) to automatically scale.
- C client aggressively reconnects in an opaque way - static 1s backoff (this happens in certain cases, not all connection losses)

etcd Pros

- HTTP protocol (with HTTPS support) - requires following HTTP redirects for writes
- Allows "dirty" reads when the cluster is leader-less.
- HTTPS allows you to keep the backend chatter secure very easily (performance is likely limited, as Go TLS support is not known to be efficient)
- RAFT algorithm is relatively well specified and somewhat understandable
- Mirrors Chubby and POSIX - either you are a directory, or you are a file and you cannot convert between types without deleting a node first.
- Simple code is relatively easy to understand
- Modular - build on top of go-raft, so storage and frontend rpc protocol are separate and presumably swappable. Extensions like SASL would be relatively easy to add to the system.

etcd cons

- HTTP protocol (lots of overhead per request, but does support pipelining)
- Leader-less writes fail rather than block indefinitely (Feels like a bug, at the very least a timeout should be used since there are likely to be transient states where there is no leader. I observed this behavior, but did not audit the relevant code.)
- JSON encoding of all responses is tiresome for binary data - it undermines the value of being able to curl data since it can't be directly used with files
- Listing "directories" returns all data in all child nodes (bug, or feature depending on how you look at it)
- leader election for application coordination is tougher to implement in a scalable way - simple approach results in thundering herd when the leader dies.
- No explicit "mkdir" command - just assumes you want to create all paths in between
- Large data likely to tickle Go garbage collection performance issues
- Large data recovery semantics unknown
- No ACL model at all right now.
- Missing tunable variables for timeouts.