Request handler

Overview

Request handler is a central component of Colibri code running on every node (client or server). Request handler takes a file operation packet (fop) and executes it by interacting with resource sub-system, authorisation and authentication mechanisms, and back-end stores. See HLD of request handler for more details.

Functional description

A fop represents a file system or storage operation. It is typically created in a response to a system call made by a user level application running on a client. After some potential re-ordering (done by NRS) a fop is queued to a request handler.

To process a fop, the request handler creates a state machine (fom) representing logic of the file system operation. The fom goes through a series of state transitions that involve updates to the data and meta-data, creation of other fops, sending messages to other nodes, waiting for messages, *&c.* Some of these state transitions are standard, in the sense that their implementation is generic for multiple fop types. Such standard state transitions (also called standard fom *phases*) include:

- authentication: verifying the capabilities with which some fop fields are signed;
- local resource acquisition: waiting until local resources (memory, cpu bandwidth and so on) are available. This phase allows control of the server load level and protects against thrashing;
- distributed resource acquisition: waiting until distributed resources (distributed locks, grants, quotas, *&c.*) are available. Request handler checks whether required resource usage rights are already granted to the local node. If not, resource requests are enqueued to the corresponding resource owners. This step involves creation of resource manager fops, their processing (again involving the request handler) and waiting until resource usage rights are granted. As a matter of policy, if some resources, either distributed or local, are not immediately available, fop execution can change its *mode* from *wbc* (write-back-cache), where fop is executed locally and its updates are cached, to intent mode, where fop is instead immediately sent to the target nodes, which own the resources. The target nodes continue the fop execution (again, selecting between wbc and intent modes). The decision to select execution mode depends on multiple factors:
    - local resource consumption on a node (intent node consumes less local resources);
    - availability of distributed resources;
    - contention on distributed resources: a server might instruct a client to not cache a particular piece of data or meta-data if this resource is contended by multiple clients;

- object checking and loading: request handler check the existence (or non-existence, as required) of local file system objects referenced by the fop and loads objects meta-data. For example, for a CREATE operation, the parent directory must exist and newly created child object—must not;

- authorisation: at this phase, request handler checks that operations specified by the fop are authorised. This is done by (i) querying user data-base to translate user identifiers into internal form, (ii) fetching authorisation data (acls) from the meta-data store and (iii) checking user identifiers against acls;

- local transaction context: updates to dara and meta-data performed as part of the file system operation are executed in a context of local transaction. Transaction context is created by the request handler. This step might involve waiting until enough space is available in the log;

- execution of a fom might involve sending other fops out, for example, sending reply fops from the server back to the client. Request handler deals with the generic part of fop sending, including waiting for a space in the local fop cache.

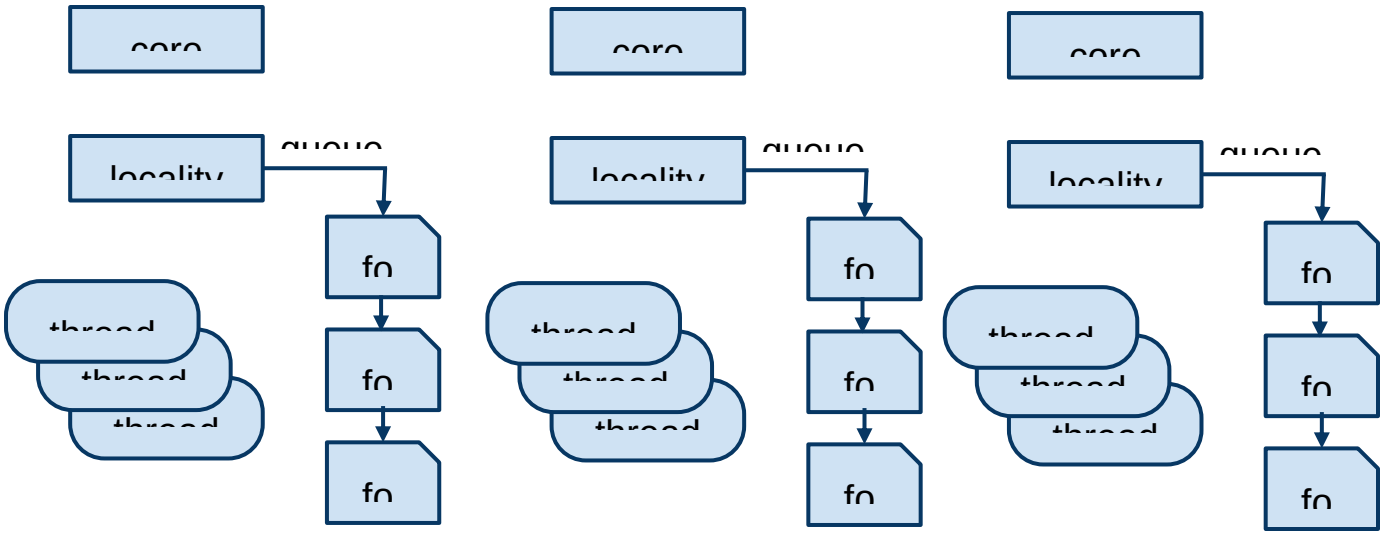
Logical description

Internal design of request handler is determined to a large degree by the following considerations:

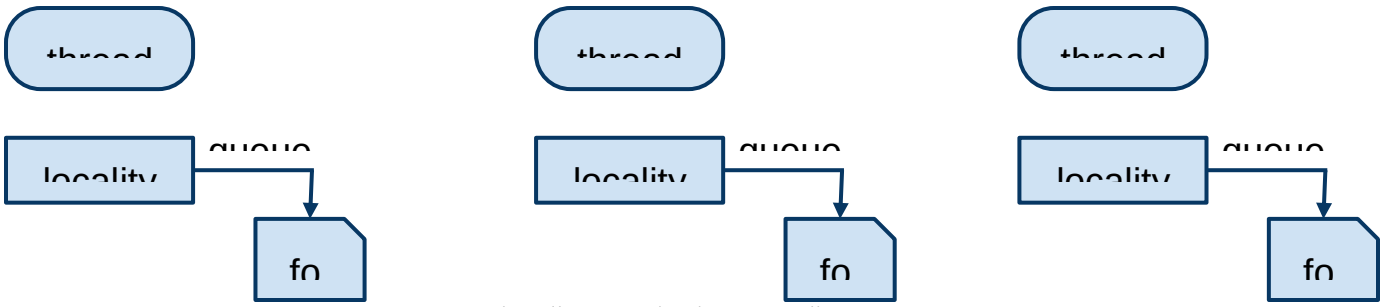
- the same request handler code should execute on a client (typically a Linux kernel module) and a server (typically a user space program);

- server-side request handler should not use massively threaded (thread-per-request) model, because of its overhead.

The solution (presented in the HLD, referenced above) is to use non-blocking model for fop execution. File operation is broken up into a series of state transitions that do not involve long waits. The resulting state machine (fom) is not bound to any particular thread and can migrate from thread to thread between state transitions. On a server, fom-s are executed by handler threads, attached to a locality, which is typically associated with a processor core. On a client, there is a locality with a single handler thread for each host thread:



Request handler organization on a server.



Request handler organization on a client

This organisation achieves the goals indicated above by detaching fom processing from locality configuration: on a server a set of cooperating threads execute state transitions of fom-s from the locality queue. On a client, when a thread enters a Colibri code (as a result of file-system system call), a locality is created with this thread as the only handler thread. The fom, representing the system call is created and executed by this thread.

Use cases

[client query cached]

An application invokes read-only system call (*e.g.*, stat(2) or read(2)) for data or meta-data already present in the local client cache.

- A fom is created for the call;
- a locality to execute this fom is created (on the stack), using the current thread as its only handler thread;
- standard phases are executed. Resource acquisition phase finds that all necessary resource usage rights are already granted;
- fop-type specific phase packs the result in user space expected format and returns it to the user space;
- fom processing completes, locality and fom are recycled.

[client query wbc]

An application invokes a read-only system call for data or meta-data not present in the local client cache.

- A fom is created for the call;
- a locality to execute this fom is created (on the stack), using the current thread as its only handler thread;
- standard phases are executed. Resource acquisition phase finds that some necessary resource usage rights are not granted. According to the request handler policy the fop should be executed in the wbc mode;
- resource acquisition requests are enqueued. When resources are granted, they are cached locally. The fop execution proceeds through the standard phases;
- fop-type specific phase packs the result (present in the local cache by this time) and returns it to the user space;
- fom processing completes, locality and fom are recycled.

[client intent]

An application invokes a read-only or read-write system call for data or meta-data not present in the local client cache.

- A fom is created for the call;
- a locality to execute this fom is created (on the stack), using the current thread as its only handler thread;
- standard phases are executed. Resource acquisition phase finds that some necessary resource usage rights are not granted. According to the request handler policy the fop should be executed in the intent mode;
- a fop representing the fom is built and sent to the corresponding nodes. This operation can be blocked for various reasons (priorities, rpc cache size limits, network bandwidth caps, *&c.*).
- the fop is executed by the server (see the corresponding use case below);
- the reply fop is sent back to the client;
- the reply is received by the client;
- the results are copied from the reply to the user space;
- fom processing completes, the locality and the fom are recycled.

[server]

A server received an incoming fop from a client.

- A fom is created;
- home locality is assigned to the fom;
- the fom is placed in the home locality run-queue;
- standard phases are executed for the fom;
- fop-type specific phases are executed. This involves calls to the underlying data and meta-data stores to query and modify file system;
- reply fop is sent to the client;
- fom processing terminates, the fom is recycled.