High level design of M0 request handler

By Nikita Danilov <nikita.danilov@clusterstor.com>
Date: 2010/04/29
Revision: 0.1

[Text in square brackets and with a light-green background is a commentary explaining the structure of a design document.]

This document presents a high level design (HLD) of M0 request handler (reqh) component. The main purposes of this document are: (i) to be inspected by M0 architects and peer designers to ascertain that high level design is aligned with M0 architecture and other designs, and contains no defects, (ii) to be a source of material for Active Reviews of Intermediate Design (ARID) and detailed level design (DLD) of the same component, (iii) to serve as a design reference document.

The intended audience of this document consists of M0 customers, architects, designers and developers.

0. Introduction

[This section succinctly introduces the subject matter of the design. 1--2 paragraphs.]

Request handler (reqh) is a M0 module containing file system (and storage) operations logic. reqh accepts a file operation packet (FOP) and interprets it by interacting with other sub-systems (resource management framework, distributed transaction manager, storage objects and containers) to execute file system operation.

reqh is a part of a M0 server as well as a M0 client.

1. Definitions

[Definitions of terms and concepts used by the design go here. The definitions must be as precise as possible. References to the M0 Glossary are permitted and encouraged. Agreed upon terminology should be incorporated in the glossary.]

- *file operation*: an update to or a query of file system state that maintains file system state consistency. Examples:
  - `mkdir("d");`
  - `rename("a/b", "c/d");`
  - `write(fd, buf, count);`
  - `stat("f");`
  - `read(fd, buf, count);`
- *file operation type*: a class of file operations, such that a particular operation from the class can be identified by a set of parameters. Examples:
  - mkdir: parameters include parent directory, new directory name, new directory permission bits, timestamps of a new directory;
  - rename;
  - write;
- *file operation packet*: a description of file operation suitable for sending over network and storing on a storage device. File operation packet (FOP) identifies file operation type and operation parameters;
- *file system object*: a file, a directory or a storage object. A file operation updates state of some file system objects. File system objects are identified by a globally unique *file identifier* (fid).

2. Requirements

[This section enumerates requirements collected and reviewed at the Requirements Analysis (RA) and Requirements Inspection (RI) phases of development. References to the appropriate RA and RI

documents should go here. In addition this section lists architecture level requirements for the component from the Summary requirements table and appropriate architecture documentation.]

- [r.reqh.extensible]: new file operation types can be added without changing core reqh code;
- [r.reqh.nrs]: reqh can collaborate with NRS to optimize resource utilization;
- [r.reqh.async]: reqh support non-blocking non-threaded computation model;
- [r.reqh.client-server]: the same reqh code runs on both client and server;
- [r.reqh.user-kernel]: the same reqh code runs in user and kernel space.

There are no special Summary requirements for this component.

3. Design highlights

[This section briefly summarises key design decisions that are important for understanding of the functional and logical specifications and enumerates topics that a reader is advised to pay special attention to.]

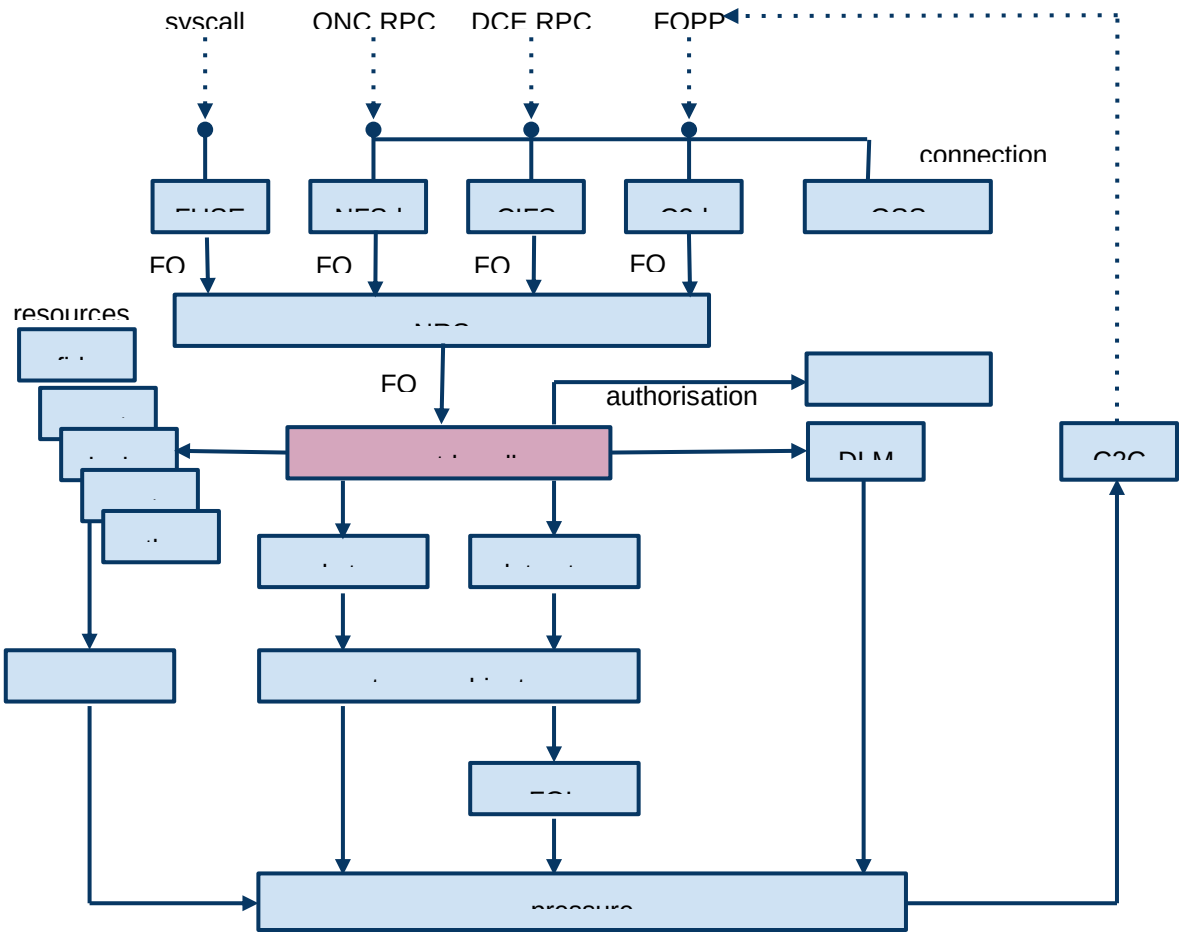
To achieve the requirement of extensibility, reqh delegates file system operation execution to *file system operation machines* (FOMs), that can be added or removed dynamically.

The intention of the design is to factor as much of functionality as possible out into generic reqh code to make file operation type specific code simple and idiomatic.

4. Functional specification

[This section defines a functional structure of the designed component: the decomposition showing *what* the component does to address the requirements.]

The following diagram illustrates the relationships between reqh and other M0 components.



reqh receives a stream of FOPs as its input. For each FOP reqh performs some standard actions:

- authenticity checks: reqh verifies that protected state in FOP is authentic. Various bits of information in M0 are protected by a cryptographic signature made by a node that issued[1] this information:
  - object identifiers (including container identifiers and fids);
  - capabilities[2];
  - locks;
  - layout identifiers;
  - other resources identifiers;
  - *etc.*

[1][u.security.originating-node]

[2][u.SECURITY.CAPABILITIES] ST

reqh verifies authenticity of such information by fetching[3] corresponding node keys[4], re-computing the signature locally and checking it with one in the FOP.

- resource limits: reqh estimates local resources (memory, cpu cycles, storage and network bandwidths) necessary for operation execution. The execution of operation is delayed[5] if it would overload the server or exhaust resource quotas associated with operation source (client, group of clients, user, group of users, job, *etc.*);
- resource usage and conflict resolution: reqh determines what distributed resources will be consumed by the operation[6] execution and call resource management infrastructure to request the resources and deal with resource usage conflicts (by calling DLM if necessary);
- object existence: reqh extracts identities of file system objects[7] affected by the FOP and requests appropriate stores to load object representations together with their basic attributes;
- authorization control: reqh extracts the identity of a user[8] (or users) on whose behalf the operation is executed. reqh then uses enterprise user data base to map user identities into internal form[9]. Resulting internal user identifiers are matched against protection and authorization information stored in the file system objects (loaded on the previous step);
- distributed transactions: for operations mutating file system state, reqh sets up local transaction context where the rest of the operation is executed.

After standard actions, independent of the file operation type have been executed, reqh calls FOM that executes file system operation in the established context. As FOM executes file operation, it queries or updates state in the stores, possibly forming and queuing one or more reply FOPs.

In addition to this, reqh collaborates with the Network Request Scheduler to re-order incoming FOP stream.

5. Logical specification

[This section defines a logical structure of the designed component: the decomposition showing *how* the functional specification is met. Subcomponents and diagrams of their interrelations should go in this section.]

---

[3][u.security.key.fetch]

[4][u.SECURITY.KEYS] ST

[5][u.resource.wait]

[6][u.fop.resource-list]

[7][u.fop.object-list]

[8][u.fop.user-list]

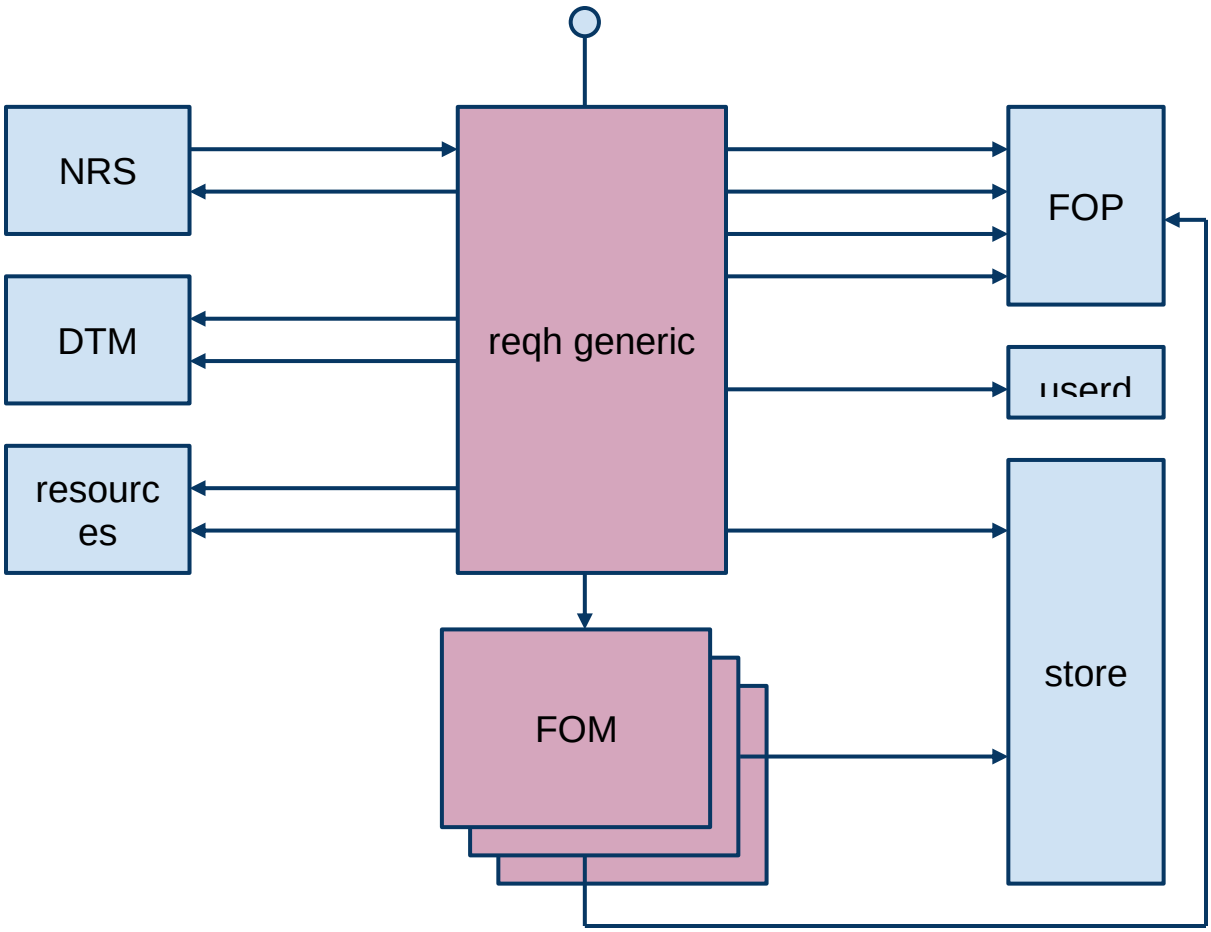
[9][u.userdb.map]

reqh consists of two major sub-components:

- generic code, that carries out common parts of file operations and
- multiple FOMs that execute file operations of a specific file operation type.

Generic code provides external interfaces to NRS in addition to its main interface through which FOPs are submitted for execution.

FOM instance encapsulates the state of file operation execution. A FOM is not bound to a thread.



5.1. Conformance

[For every requirement in the Requirements section, this sub-section explicitly describes how the requirement is discharged by the design. This section is part of a requirements tracking mechanism, so it should be formatted in some way suitable for (semi-)automatic processing.]

- [r.reqh.extensible]: reqh generic code and FOM interact through a well-defined interface. New FOP type can be registered dynamically and FOMs of this type would be created when

matching FOPs arrive;
- [r.reqh.nrs]: NRS call reqh to obtain a sort-key of a FOP. Incoming FOPs are sorted by this key and executed in the order determined by sort-key;
- [r.reqh.async]: FOM encapsulates the whole file operation execution state. All external functions called by reqh generic code and FOMs are non-blocking (in the sense, that they do not involve explicit network or storage operations; they still can block implicitly, *e.g.*, as a result of memory allocation). When reqh reaches a point where file operation execution cannot proceed until some event happens (IO completion, resource availability, *etc.*), all file operation execution state is stored in FOM and the latter is *de-scheduled* from the thread. The thread continues with the execution of next incoming FOP;
- [r.reqh.client-server]: the same reqh code runs on both client and server. See use cases section for description of how this requirement is discharged;
- [r.reqh.user-kernel]: the same reqh code runs in user and kernel space. reqh uses very simple locking (possible thanks to non-blocking FOMs) and a small amount of stack space, making it possible to run reqh code in a constrained kernel environment.

5.2. Dependencies

[This sub-section enumerates other system and external components the component depends on. For every dependency a type of the dependency (uses, generalizes, *etc.*) must be specified together with the particular properties (requirements, invariants) the design depends upon. This section is part of a requirements tracking mechanism.]

- fop
  - [r.fop.object-list]: it is possible to obtain a list of file system objects affected by a given FOP;
  - [r.fop.user-list]: it is possible to obtain a list of users on whose behalf operation is executed;
  - [r.fop.resource-list]: it is possible to obtain a list of resources required for the FOP execution;
- enterprise user data base
  - [r.userdb.maps]: user data base maps external user identities into internal form;
- security sub-system
  - [r.security.key.fetch]: node and object keys can be fetched from the issuing node;
  - [r.security.originating-node]: it is possible to determine the node that originated a given protected state;
- resources
  - [r.resource.wait]: it is possible to wait until the resource is available.

5.3. Security model

[The security model, if any, is described here.]

As described in the Functional specification section, reqh validates cryptographic signatures of protected state before using this state for file operation execution.

5.4. Refinement

[This sub-section enumerates design level requirements introduces by the design. These requirements

are used as input requirements for the detailed level design of the component. This sub-section is part of a requirements tracking mechanism.]

- resource limits must be enforced *before* any resource is consumed by the operation execution.
- a FOM runs in a transactional context

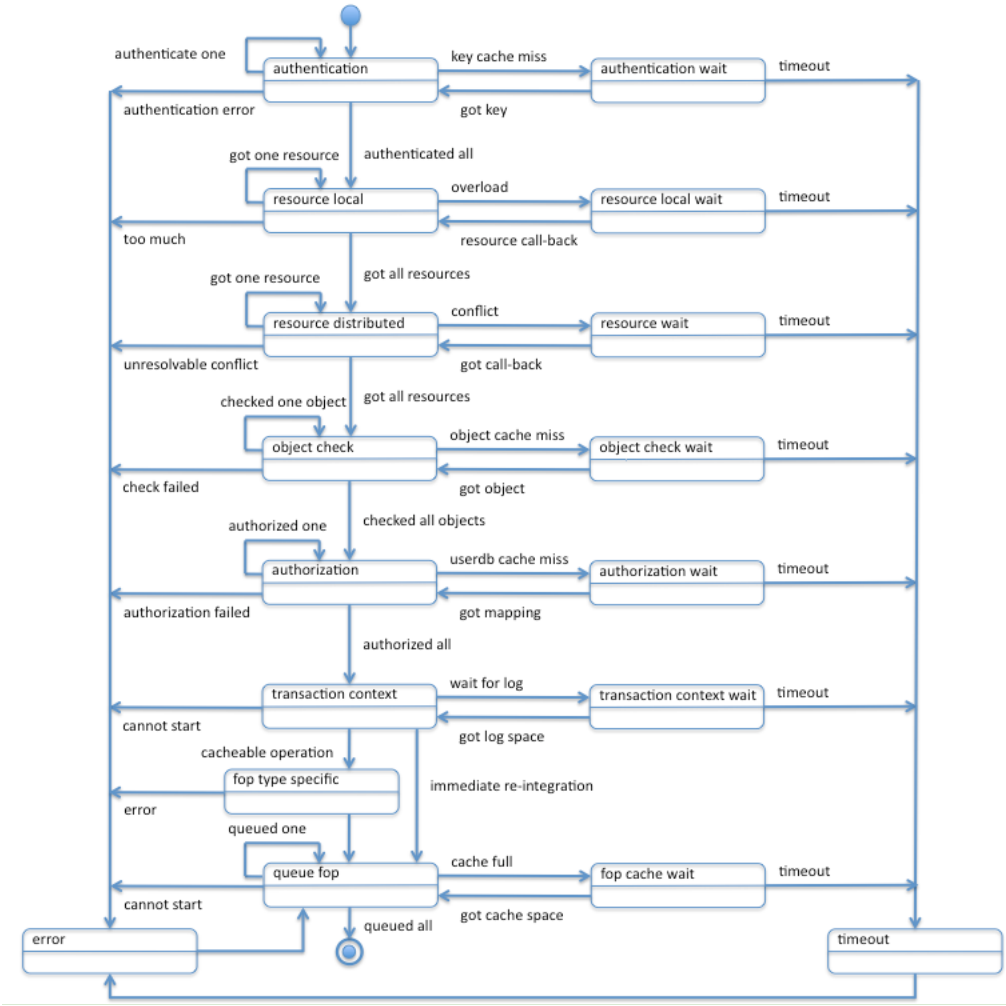
6. State

[This section describes the additions or modifications to the system state (persistent, volatile) introduced by the component. As much of component behavior from the logical specification should be described as state machines as possible. The following sub-sections are repeated for every state machine.]

Almost all state maintained by reqh is stored in FOM instances of currently executing operations. Part of FOM state has the same structure for all types of FOMs. In addition to this common state, each type of FOM (*i.e.*, each file operation type) has state specific to it. This section describes common FOM state.

6.1. States, events, transitions

[This sub-section enumerates state machine states, input and output events and state transitions incurred by the events with a table or diagram of possible state transitions. UML state diagrams can be used here.]



6.2. State invariants

[This sub-section describes relations between parts of the state invariant through the state modifications.]

6.3. Concurrency control

[This sub-section describes what forms of concurrent access are possible and what forms on concurrency control (locking, queuing, *etc.*) are used to maintain consistency.]

On a server reqh creates a number of service threads (perhaps a fixed number of threads per processor). Each of these threads takes incoming FOP from a queue and creates a FOM for it. If FOM blocks waiting for an event (see the diagram above), service thread switches to the next available FOM. A FOM can be processed by no more than one service thread at a time, making locking unnecessary.

On a client, in addition to service threads, reqh code is executed by the system call threads (assuming in-kernel client). The same invariant is maintained: a FOM can not be concurrently processed by multiple threads.

7. Use cases

[This section describes how the component interacts with rest of the system and with the outside world.]

7.1. Scenarios

[This sub-section enumerates important use cases (to be later used as seed scenarios for ARID) and describes them in terms of logical specification.]

Scenario	[usecase.reqh.client.caching]
Relevant quality attributes	usability, flexibility
Stimulus	system call on a client
Stimulus source	user application
Environment	M0 file system mounted on a client node
Artifact	FOP created by the file system entry point
Response	reqh generic code running on the client creates a FOM for the FOP and runs it through the common state transitions as described above. Some of the phases might be simpler (or omitted altogether) on a client. For example, authentication is probably not needed.
Response measure	During distributed resource and object checking phases, FOM decides whether it has to be cached or it has to be forwarded to the server immediately. The decision depends on many factors: ● availability of lock and other resources on the client; ● contention on the resources (caching is counter-productive for the highly contentious resources, as it results in the lock ping-pong); ● memory pressure on the client, *etc.* If caching is preferable, the FOP type specific part of file operation is executed over local objects. As part of execution in the local transactional context, records are added to the client FOL. Later state updates will be re-integrated to the server.
Questions and issues	

Scenario	[usecase.reqh.client.notcaching]
Relevant quality attributes	usability, flexibility
Stimulus	system call on a client
Stimulus source	user application
Environment	M0 file system mounted on a client
Artifact	FOP created by the file system entry point

Response	the same as in [usecase.reqh.client.caching]
Response measure	If immediate execution is preferable, FOP type specific part of file operation is not executed locally. Instead, the FOP is marked as non-cacheable (urgent) and will be sent to the server for re-integration as soon as possible.
Questions and issues	

Scenario	[usecase.reqh.server]
Relevant quality attributes	usability, flexibility
Stimulus	an incoming RPC on a M0 server
Stimulus source	RPC sent to the server from a client
Environment	normal operation of a M0 file system
Artifact	FOP is created by the server protocol translator and submitted to reqh for execution
Response	reqh creates FOM and executes is until completion as described above
Response measure	FOM execution is non-blocking allowing reqh with a relatively few threads. After common FOM phases and fop type specific phases are executed, reqh queues a number of reply FOPs to be sent back to a client (and potentially to other nodes)
Questions and issues	

[UML use case diagram can be used to describe a use case.]

7.2. Failures

[This sub-section defines relevant failures and reaction to them. Invariants maintained across the failures must be clearly stated. Reaction to Byzantine failures (*i.e.*, failures where a compromised component acts to invalidate system integrity) is described here.]

Execution of an FOM is arranged so that it prepares all the resources that might be required before it updates any file system state: objects are loaded, resources are acquired and locks are taken before stores are updated. This simplifies failure handling, because there is nothing to undo when any of these preliminary phases fails.

FOP type specific actions are executed in the transactional context, simplifying failure handling—a transaction can be undone as a whole.

Byzantine clients are dealt with by a signature based authentication phase.

Other types are failures (node, network, storage, *etc.*) are outside of the scope of this document.

8. Analysis

8.1. Scalability

[This sub-section describes how the component reacts to the variation in input and configuration parameters: number of nodes, threads, requests, locks, utilization of resources (processor cycles, network and storage bandwidth, caches), *etc.* Configuration and work-load parameters affecting component behavior must be specified here.]

It is believed that significant improvements in the throughput (in terms of a number of FOPs processed per second) can be obtained by binding FOP processing to a physical processor (or, rather, to a cache). reqh with a small number of processor-bound threads handling incoming FOPs in a non-blocking fashion is designed to take advantage of this. Additionally, non thread-based reqh decreases the cache footprint (by eliminating per-thread stacks) and decreases the number of context switches.

8.2. Other

[As applicable, this sub-section analyses other aspects of the design, *e.g.*, recoverability of a distributed state consistency, concurrency control issues.]

8.2. Rationale

[This sub-section describes why particular design was selected; what alternatives (alternative designs and variations of the design) were considered and rejected.]

9. Deployment

9.1. Compatibility

[Backward and forward compatibility issues are discussed here. Changes in system invariants (event ordering, failure modes, *etc.*)]

9.1.1. Network

9.1.2. Persistent storage

9.1.3. Core

[Interface changes. Changes to shared in-core data structures.]

9.2. Installation

[How the component is delivered and installed.]

10. References

[References to all external documents (specifications, architecture and requirements documents, *etc.*) are placed here. The rest of the document cites references from this section. Use Google Docs bookmarks to link to the references from the main text.]

11. Inspection process data

[The tables below are filled in by design inspectors and reviewers. Measurements and defects are transferred to the appropriate project data-bases as necessary.]

11.1. Logt

	Task	Phase	Part	Date	Planned time (min.)	Actual time (min.)	Comments
umka		HLDINSP	1	2010/5/12	96	30	1
jay		HLDINSP	1		96		

11.2. Logd

No.	Task	Summary	Reported by	Date reported	Comments
1		nothing is said about barriers to be implemented with special fop by nrs + reqh. This is not defect because strictly speaking they belong to nrs arch	umka	2010/5/12	
2					
3					
4					
5					
6					

7					
8					
9					