DLD of Parallel Lock Callback

Niu YaWei

2006-12-6

1 Functional Specification

1.1 int ldlm_run_bl_ast_work(struct list_head *rpc_list)

This function will be modified. It'll send the blocking callbacks(of the rpc_list) in parallel and wait for all the callbacks done.

1.2 int ldlm_run_cp_ast_work(struct list_head *rpc_list)

This function will be modified. It'll send the completion callbacks(of the rpc_list) in parallel and wait for all the callbacks done.

1.3 int ldlm_server_blocking_ast(struct ldlm_lock *lock, struct ldlm_lock_desc *desc, void *data, int flag)

This function will be modified. It'll just prepare a request for the blocking callback then add the request to a request set. The request set is passed in by the "data" argument.

1.4 int ldlm_server_completion_ast(struct ldlm_lock *lock, int flags, void *data)

This function will be modified. It'll just prepare a request for the completion callback then add the request to a request set. The request set is passed in by the "data" argument.

1.5 static int ldlm_cb_interpret(struct ptlrpc_request *request, void *data, int rc)

This is a new reply interpret function. It'll handle the error of blocking/completion callback, then propagate the -ERESTART error by use of the "data" if necessary.

2 Use Cases

2.1 sending blocking callbacks when try to enqueue a conflicting lock.

- The server enqueue thread checks conflicts and generates the conflicting lock list, then it calls ldlm_run_bl_ast_work() with the argument of the conflicting lock list to send blocking callbacks.
- ldlm_run_bl_ast_work() creates a request set, then it calls the ldlm_server_blocking_ast() for each lock in the blocking list, at last, it calls ptlrpc_set_wait() to wait for the request set done.
- ldlm_server_blocking_ast() prepares a request for sending the blocking callback, then it adds the request into the request set.
- The ptlrpcd thread sends all the requests of request set in parallel.
- ldlm_cb_interpret() is called to handle the error of blocking callback when receiving the callback reply(or the sending ErrorReturn).
- The enqueue thread is waken up after the request set done.

2.2 sending completion callbacks when reprocess waiting list.

- The server cancel thread reprocess the waiting list after a lock cancellation and generates the completion lock list, then it calls ldlm_run_cp_ast_work() with the argument of the completion list to send completion callbacks.
- ldlm_run_cp_ast_work() creates a request set, then it calls the ldlm_server_completion_ast() for each lock in the completion list, at last, it calls ptlrpc_set_wait() to wait for the request set done.
- ldlm_server_completion_ast() prepares a request for sending completion callback, then it adds the request into the request set.
- The ptlrpcd thread sends all the requests of the request set in parallel.
- ldlm_cb_interpret() is called to handle the error of completion callback when receiving the callback reply(or the sending ErrorReturn).
- The cancel thread is waken up after the request set done.

3 Logic Specification

3.1 int ldlm_run_bl_ast_work(struct list_head *rpc_list)

```
struct ldlm_cb_set_arg {
```

```
    ptlrpc_request_set *set;
    atomic_t           restart;
    unsigned           short type;  /* LDLM_BL_CALLBACK or LDLM_CP_CALLBACK) */
}
int ldlm_run_bl_ast_work(struct list_head *rpc_list)
{
    struct ldlm_cb_set_arg arg;
    arg.set = ptlrpc_prep_set();
    atomic_set(&arg.restart, 0);
    arg.type = LDLM_BL_CALLBACK;
    list_for_each_safe(tmp, pos, rpc_list) {
        struct ldlm_lock *lock = list_entry(tmp, struct ldlm_lock, l_bl_ast);
        ...
        rc = lock->l_blocking_ast(lock, &desc, (void*)&arg, LDLM_CB_BLOCKING);
    }
    rc = ptlrpc_set_wait(arg.set);
    ptlrpc_set_destroy(arg.set);

    return(atomic_read(arg.restart) ? -ERESTART : 0);
}
```

3.2 int ldlm_run_cp_ast_work(struct list_head *rpc_list)

```
int ldlm_run_cp_ast_work(struct list_head *rpc_list)
{
    struct ldlm_cb_set_arg arg;
    arg.set = ptlrpc_prep_set();
    atomic_set(&arg.restart, 0);
    arg.type = LDLM_CP_CALLBACK;

    list_for_each_safe(tmp, pos, rpc_list) {
        struct ldlm_lock *lock = list_entry(tmp, struct ldlm_lock, l_cp_ast);

        ...

        rc = lock->l_completion_ast(lock, 0, (void*)&arg);
    }
    rc = ptlrpc_set_wait(arg.set);
    ptlrpc_set_destroy(arg.set);

    return(atomic_read(&arg.restart) ? -ERESTART : 0);
}
```

3.3 int ldlm_server_blocking_ast(struct ldlm_lock *lock, struct ldlm_lock_desc *desc, void *data, int flag)

```
int ldlm_server_blocking_ast(struct ldlm_lock *lock, struct ldlm_lock_desc *desc, void
{
    struct ldlm_cb_set_arg *arg = (struct ldlm_cb_set_arg *)data;
    ptlrpc_request *req = ptlrpc_prep_request(...);
    req->rq_async_args.u.pointer_arg[0] = arg;
    req->rq_interpret_reply = ldlm_cb_interpret;

    ...

    if (unlikely(instant_cancel)) {
        rc = ptlrpc_send_rpc(req, 1);
        atomic_inc(&arg->restart);
    }
    else {
        rc = ptlrpc_set_add_req(arg->set, req);
    }
    return rc;
}
```

3.4 int ldlm_server_completion_ast(struct ldlm_lock *lock, int flags, void *data)

```
int ldlm_server_completion_ast(struct ldlm_lock *lock, int flags, void *data)
{
    struct ldlm_cb_set_arg *arg = (struct ldlm_cb_set_arg *)data;
    ptlrpc_request *req = ptlrpc_prep_request(...);
    req->rq_async_args.u.pointer_arg[0] = arg;
    req->rq_interpret_reply = ldlm_cb_interpret;

    ...

    rc = ptlrpc_set_add_req(arg->set, req);
    if (instant_cancel)
        atomic_inc(&arg->restart);
    return rc;
}
```

3.5 static int ldlm_cb_interpret(struct ptlrpc_request *request, void *data, int rc)

```
static int ldlm_cb_interpret(struct ptlrpc_request *request, void *data, int rc)
{
    struct ldlm_cb_set_arg *arg = data;

    ptlrpc_req_finished(req);
    if (rc != 0)
        rc = ldlm_handle_ast_error(lock, req, rc, arg->type == LDLM_BL_CALLBACK ? "bloc
    if (rc == -ERESTART)
        atomic_inc(&arg->restart);
    return 0;
}
```

4 State Specification

4.1 Resources Involved and Their State

In current LDLM, the lock enqueue thread may stall on sending blocking/completion callbacks when there is a huge blocking/completion list, since it always sends the callback RPCs one by one. With the new parallel callback mechanism, the lock enqueue thread will just add each request into a request set then go to sleep, ptlrpcd will manage to send the RPCs in the request set simultaneously, after all requests done, ptlrpcd wake up the lock enqueue thread to move on.

4.2 Locking

Using resource lock (lr_lock) carefully to protect the lists/flags of lock and resource.

4.3 Recovery

No effect.

5 Environment

5.1 Network Protocol Compatibility - New clients, Old Servers

None.

5.2 Network Protocol Compatibility - Old clients, New Servers

None.

5.3 Disk Format Changes

None.

5.4 Documentation Changes

None.