Server

Clustering with servers. Serivces are core of the cluster. Node is named as server since it is old good friend. Connection losts, service crashes, instability of nodes and services are also bad old friends. Instead of trying to solve these issues at framework, actors using services will process problems directly subscribing to those bad events. Application can handle issues better considering its context and state.

server

- starts net::service
- listens on protocols
- connects to peers
    - send server information
    - fixed id
- starts services from json
    - using task_adaptor
- starts a task_scheduler
- manages service_directory
    - reference to service
        - local or remote
        - acquire
            - service::ref like session::ref
            - sub to state
    - state update
- not a singleton
    - to test two servers in a unittest

service

- an actor
- id
    - node.actor id
- role
    - string
    - examples: "login", "match", "lobby", "instance_runner"
- msg_service_base
    - id
    - msg_service_state

        - up / down
        - propery : json string
            - parsed when received
- that's all

Using local and remote service

- service::ref
    - local or remote
    - kept in service_directory
    - send
    - state subscription
- recv
    - follow usual "session" channel subscription
    - group to message type mapping
        - service: msg_service casting with groups
    - group / type to message type mapping
        - usual subscription inside service
    - service:
        - topic / address scheme can work for all dispatching
    - client:
        - topic
        - session / address mapping required for security reason

Code - Dispatch msg_service

```
// server:
auto sch = channel::find("session");
sch.subscribe( topic(61, 0), dispatch_with_address, immediate );
...
sch.subscribe( topic(99, 0), dispatch_with_address, immediate );

// range subscription can be easier
sub.subscribe_groups( 61, 99, dispatch_with_address, immediate)

// dispatch_with_address:
auto sm = std::static_pointer_cast<msg_service>(m);
auto scv_channel = find_service(ms->dst[0]).get_channel();
svc_channel->push(sm);
```

Code - Dipsatch msg_client

```
// server dispatch to service with groups, then each service dispatches.
// example, instance_runner_service to user actor

channel_.subscribe( topic(101, 0), dipsatch_with_session_instance, immediate );

// dispatch_with_session_instance
auto um = std::static_pointer_cast<msg_user>();
um->user_id = get_user_id_from_session(um->sid);

auto inst = get_user_instance_from_user_id(um->user_id);
inst.post(um);

// instance posts to target users
// - for even distribution of processing time for instances
// - if all messages are processed here, update of instances can be delayed
```

Code - Joining one of instance_runner_service

```
// client needs to have context, and content based dispatching
// lobby_service can be used to dispatch based on context and content

// lobby_servcie:
channel("session").subscribe( topic(101, 0), on_join);

// on_join:
auto msg_join = static_pointer_cast<msg_join_instance>(m);

if ( user can join the instance )
{
  msg_join->user_id = get_user_id_from_session(m->sid);
  msg_join->... = ... // more data for instance_runner_service

  server_.find( target instance address ).channel().push( m );
}

//
// after initial verification, instance_runner_service can process
// other messages directly with subscription to channel("session")
//
```

Examples

Instance Architecture

A instance oriented game server. Well known examples are FPS, MORPG, board games, and MOBA games. Most games are instance based except some MMORPGs providing seamless world. But even those MMORPGs can be developed with instances if smooth instance transition is possible under game design.

Server and Services

- front servers running:
  - instance_runner_service(s)
    - runs instance actors of different worlds
  - lobby_service
    - keep users
    - relay backend messages
  - login_service
- world backend servers running:
  - world_service
    - authorize
    - keeps users
    - relays world traffic
  - guild_service
  - instance_control_service
  - world / guild subs to user state messages
    - copied user state
- master backend server running:
  - master_service
    - controls services
      - start / stop services on front servers

Stability

world backend services are failure points

- recovery process
  - notify users on lobby_service joined the world
  - keep each instance of the world in suspended mode
  - restart the world server
  - when services of the world are restarting
    - notify users
    - update state of user, guild, instance
  - when services are up
    - notify users
    - resume instances

Match Making Service

1 billion match making.

- match_service
  - 100K match processing per service
  - dynamic partition of ranges with transition
- range partition
  - when flooded
  - $(l, u) \rightarrow (l, m)(m, u)$
  - create a new match_service on a proper server with (m, u)
  - advertise new ranges to all related services
  - shrinking phase
    - decrease selection probability of the shrinking server
    - most requests to the new range partition go to the new service
    - timeout of matching can clear all existing matches that cannot be matched
- range merge
  - when idle for a long time
    - ask merge of partition
    - if agreed, then extend range and stop one the two

Recv Dispatching

Look at use cases carefully. Use case and usage drive design. Core relation.

1. instance creation request
2. instance join request from users
3. instance leave request from users

messages need to have destination somehow. source and destination are required to identify each other. server.actor_id is the key to identify actors on all systems. This can be used for source and destination key.

```
struct loc { uint16_t server_id; actor::id id };
struct msg_actor {
    loc src;
    loc dst;
};
```

service can provide default callback for dst based subscription. group / type based topic can provide efficient dispatching with it, but there needs to be a type based dispatching.

$type\_map : topic \rightarrow type$ is a function that maps a type from a topic. It is based on convention, not based on type system. This has been a established way since network communication removes type information from messages. This limitation (type decay) forces dynamic type system.

lax uses type_info and typeid based dynamic type system for components and actors. This can be re-used for messages and can provide type based subscription and dispatching.

```
if ( msg->is_a<msg_actor>())
{
  // cast and get src and dst
  // dispatch to subsribers for src and/or dst
}
```

Basic idea is found, but performance and usage are important.

This type system can be used to bind user id to user messages from a session.

```
if ( msg->is_a<msg_user>() )
{
  // find a user id from a session
  // set user_id of the msg
}
```

Who uses the type system to dispatch? Another check for each message is required. Server knows local and remote services. Service knows local actors. Actors are standalone execution entities. That requires hierarchical dispatching.

- 1st level dispatching (server)
  - subscribe to channel "session" for all interested group messages
  - that subscription uses address based dispatching to services
- 2nd level dispatching (service)
  - subscribe to its own channel for group or group / topic when direct subscription from actors is not possible. (when further dispatching based on message content is required)

starting from topic based sub/post, we found that address or content based dispatching is required.

It is tempting to use server.service[.actor] as address and use it to post.

- message content based dispatching
  - topic based dispatching
  - message type based dispatching
  - address based dispatching
  - session based dispatching
    - a content based dispatching when session id is in message

The all schemes are content based dispatching, a very general concept. But it can be complicated when implementing. Minimal requirements are following:

- must be easy to use
- must support topic and content based dispatching to some degree