RC25410 (WAT1309-048) September 20, 2013
Computer Science

IBM Research Report

Architecture for a Workload Centric Cloud

Mike Spreitzer, Diana Arroyo, Malgorzata Steinder
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 208
Yorktown Heights, NY 10598
USA

Architecture For A Workload Centric Cloud

Mike Spreitzer, Diana Arroyo, Malgorzata Steinder

Table of Contents

Overview

This document outlines an architecture for a Workload Centric Cloud (WCC) built on top of an extended OpenStack cloud. A Workload Centric Cloud serves primarily to deploy, manage, update, and delete Virtual Resource Topologies (VRTs). The Workload Centric Cloud accepts requests at its level of abstraction and serves them by manipulating the infrastructure APIs (Software Defined Compute, Storage, and Networking) of an underlying OpenStack+ installation. By OpenStack+ we mean OpenStack as we plan to augment it; this includes, for example, better Software Defined Networking (SDN) functionality, and extensions to improve visibility into Software Defined Compute.

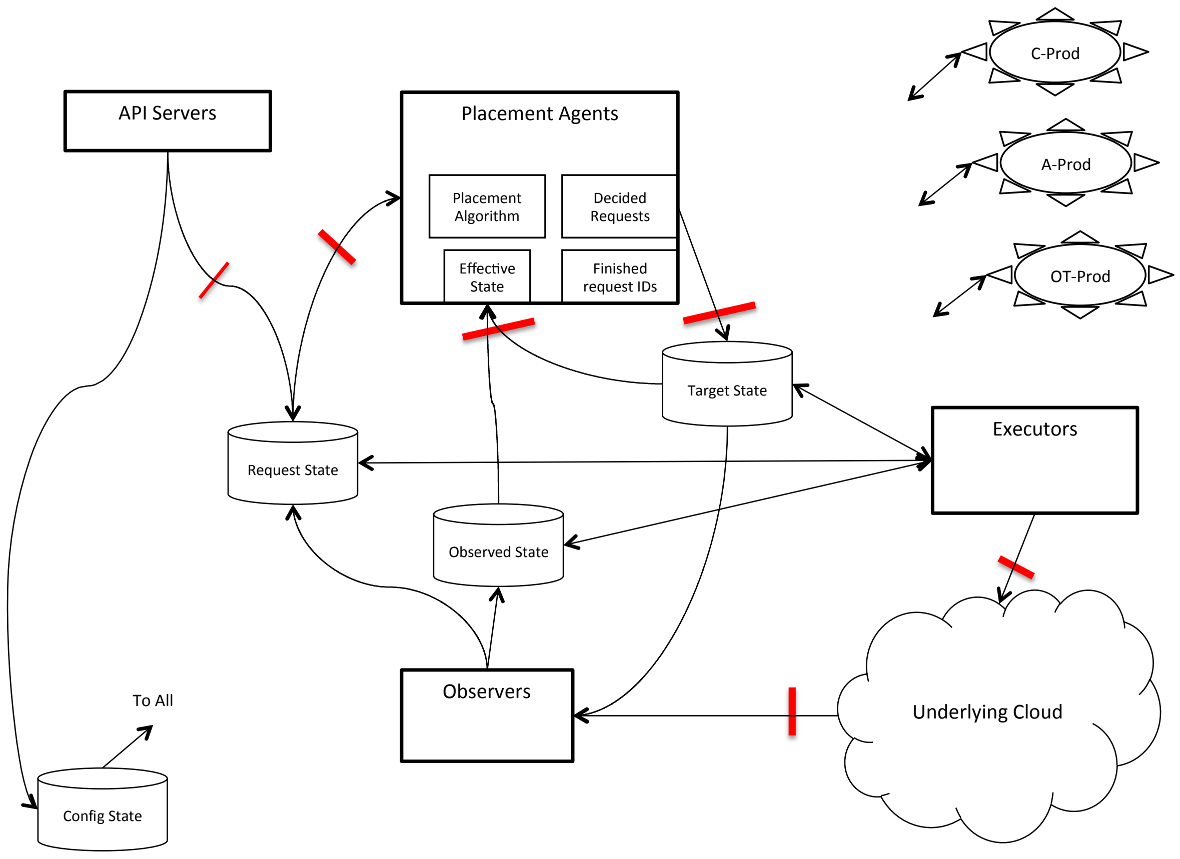

Figure 1

See Figure 1 for an overview picture. A box drawn with a heavy line represents a set of processes. An arrow shows information flow; a red bar indicates model transformation. A cylinder shows a section of database contents. A sun-shape shows a pub/sub topic.

The picture and this text focus on the handling of requests to do something. There are also requests for information, which are handled by querying the relevant database contents; all the information in the database can be queried, although some can only be queried by administrators.

Many do-something requests are handled by a collection of distinct processing steps. Steps in distinct processes are connected by logical work queues. A logical work queue is data in the database plus usage of a pub/sub topic to work around the fact that we do not have confidence in database trigger technology. “Queue” is a bit of a misnomer here because we do not use FIFO ordering. These logical work queues all have the same pattern for how work is inserted: after committing a DB transaction that inserts work the inserter publishes a prod message to alert workers and, upon crash recovery, the inserter publishes prod messages to cover (possibly making a big over-estimate) for ones that were missed due to the crash. In particular, this design has the following work "queues".

- Requests awaiting placement decisions.
- VRTs with updated target state awaiting execution and reading into placement agent effective state.
- VRTs with abandonment bits awaiting incorporation into placement agent effective state.

VRTs

A VRT is an infrastructure-level description of a collection of virtual resources that are to be deployed, managed, updated, and deleted as a unit. These virtual resources are things like VM instances, virtual storage volumes, and various sorts of networking things. By virtual here we really mean "manageable by software", not necessarily, e.g., "hosted by a hypervisor". If, e.g., OpenStack+ can do a "bare metal" deploy of a machine image, we consider that deployed image to be a VM instance.

The virtual resources in a VRT are organized into a tree of groups. Various sorts of policy statements can be attached to groups. Examples include: mutual co-location or anti-co-location constraints to be applied among all members, or declarations of the presences of certain licensed software products. Relationships between groups, and between groups and individuals, can be declared. Example relationships include network reachability, co-location or anti-co-location, network bandwidth or latency requirements, and limits on network hop count. Each policy or relationship is independently marked as a hard constraint or as a preference.

We suppose that the real containers that host the virtual resources are arranged into a hierarchy. This is admittedly an approximation, but we hope it will serve us well enough. Each co-location or anti-co-location statement about virtual resources is parameterized by a level in the physical hierarchy

Requests

(Remember that we are explicitly discussing only requests to do something.)

There are two sorts of requests, and they come from two sorts of clients. There are Virtual Resource Topology (VRT) requests and administrative requests. There are administrative clients, and regular clients.

A VRT request, which is allowed from any sort of client, asserts the desired virtual state of a VRT. The VRT has an identifier that is unique in the system, as does each of the VRT's components. The virtual state of a VRT is everything that a regular client is concerned with regarding that VRT. This includes VMs and storage volumes, and their relationships --- including networking. This does not include physical resources.

In this design we augment each VM with a bit indicating whether recovery from crash of that VM (i.e., its (late) hypervisor can not continue running the VM) can be effected by a new launch of that VM --- using the same prescription, including startup stuff, as before. For each storage volume, we presume there is no automatic crash recovery. For each network relationship, we presume restoring the relationship is always an effective recovery. This design aims to effect automatic crash recovery when no help from higher layers is needed.

An administrative request is accepted only from administrative clients and is one of these sorts:

- Quiesce or un-quiesce a given physical resource
- Set part of the configuration of WCC
- Set a fudge factor

A request is implicitly a license for WCC to make necessary changes. But what exactly does "necessary" mean? Can the system go through an arbitrary set of intermediate states to get to a final state that satisfies the request? We say "no". The allowed intermediate states are in between the current state and the final state chosen to satisfy the request. For example, if a VM is "currently" running and the request is for that VM to be running, then that VM must be running in all the intermediate states. Further, in every intermediate state that VM must run on either its "current" host or the host chosen to satisfy the request. This severe restriction is imposed for the sake of simplifying the job of the planner. This restriction rules out many solutions, and thus changes some problems from solvable to unsolvable. We accept that cost, anticipating that it will rarely appear.

Does a request to add some load imply a license to move existing load (either in the same VRT or not)? This design is intended to support such license; if and when to use that license is an available degree of freedom.

Parts of the System

The WCC implementation consists of processes, a database, and a lightweight pub/sub system. There are also processes involved in the implementation of the database, the pub/sub system, and perhaps the process monitoring involved here --- but those processes receive no further attention here.

The Database

There is one highly available database, which is divided into four sections: configuration state, request state, target state, and observed state. The database can do ACID transactions of arbitrary scope. In case of concurrent transactions that conflict, the database will detect the conflict and report it in some way that the client (WCC) can identify as indicating a concurrency conflict. In the case of pessimistic concurrency control, we are not sure how promptly we can expect the database to detect a deadlock. To avoid unnecessary challenges for the database, we aim to keep database transactions relatively small.

Configuration State

The configuration state is the runtime-variable part of the configuration of the WCC implementation (as opposed to the underlying cloud it is managing). For example, it may include the desired number of API servers, the desired number of placement agents, and so on.

Request State

The request state holds requests and some of their processing status, from the time of the request's arrival to a later time when it is migrated from active storage to archival storage (if any). The requests themselves are immutable. The fragment of a request's processing status that is kept in request state consists of these things:

- its submission time,
- its processing stage,
- the ID, if any, of the placement agent currently processing the request,
- its success/failure state,
- its VRT set.

The submission time of a request is a date & time read from the local clock of the API server accepting the request. It is used to serialize requests about the same resource. (Available freedom: should we instead make the client supply the time of the request?)

A request nominally moves through the following series of processing stages; failure recovery might involve jumping backwards to retry, or skipping ahead over moot stages.

1. **ARRIVED –** meaning that the request has been accepted into the system but not otherwise acted upon
2. **PLACING –** meaning that a placement agent is working on deciding what to do about the request and writing that decision into target state
3. **TARGETED –** meaning that the decision has been written into the target state but not yet read by all the placement agents
4. **SEMI-DONE –** meaning that all the placement agents have read the decision from the target state, *not* meaning that the decision has been fully[1] put into effect (*that* question also involves the status of the request's VRTs)
5. **FULLY-DONE –** meaning that all the placement agents have read the decision from the target state and that decision has been fully put into effect

The FULLY-DONE stage is really a special case of SEMI-DONE and does not strictly need to exist as a distinct stage. Its use is a matter of de-normalization; we use it only to make certain processing easier (the leading example is in the way a Placement Agent selects a request to process).

The success/failure state of a request indicates whether the request has failed and, if so, gives details on what went wrong (Available freedom: exactly what do those details look like?). This includes information that can be queried by regular clients and is sufficient

[1] As fully as it ever will be --- note that the Executor may give up on some parts of the decision, and this does not stop the request from reaching the SEMI-DONE or FULLY-DONE stages.

for them to recover in one way or another. There is much wiggle room here regarding what information is present. As often as we can, we want this information to be enough to enable a smarter response than simply “delete it all and start over”. We want to support automated clients.

The VRT set of a request contains the ID of each VRT that the request directly affects. A VRT request’s VRT set contains just the ID of the VRT of the request. A quiesce request’s VRT set contains the ID of each VRT that “currently” uses the physical resource to be quiesced; this set is fixed at admission time by the API server, which reads that set from the target and observed state (arrows omitted from the picture for brevity). An un-quiesce request’s VRT set is empty, as is the VRT set of a WCC configuration or fudge factor change request.

The request state also includes a set of *TSR records* (TSR stands for Target State Read). The key of a TSR record consists of a request ID and a placement agent ID, and the record has no additional information (except possibly some debug information). The existence of a TSR record indicates that the identified placement agent has read the target state updates that were made for the identified request. A placement agent creates a TSR record as it does that read, and these records are deleted when the request enters the SEMI-DONE or FULLY-DONE stage. This set enables detection of whether a request’s updates have been read into a given placement agent’s effective state, which in turn serves the purpose of serializing decision making for a given VRT or physical resource; finding an alternate solution is harder than you might think.

The request state also includes a table with some data for each VRT. A VRT’s data includes a stage drawn from the following set:

1. **NEEDY –** meaning that changes have been written into the target state but no executor is working on those changes
2. **EXECUTING –** meaning that an executor is working on this VRT
3. **DONE –** meaning that no more work is needed on this VRT

Also associated with a VRT is the ID, if any, of the executor currently working on that VRT (at most one executor at a time works on any given VRT). A VRT also has a success/failure state, analogous to that of a request; this records the outcome of the latest attempt to execute a decision for that VRT.

A VRT also has a set of hard constraint violations that arose after execution. Violations are put into the set by the Observers, and removed when a new solution is written into the target state. This information should be made available to clients.

Also associated with a VRT is a set of request IDs, called the *targeted* requests. These are the requests for which a decision about the VRT has been written into the target state but not yet executed in the underlying cloud. By consulting this set it is possible to tell whether a request in the SEMI-DONE stage has been fully processed.

The request state also includes *abandonment bits,* which records the success or failure of the Executors in pushing the target state into the true state. For every piece of target observable state (see definition below), there is a bit in the abandonment state (AS) indicating whether the Executors have (1) succeeded or (2) given up on pushing that piece of TOS into the true state. When target state is created/updated by a Placement Agent: it presumes that the Executors will succeed, and sets the relevant abandonment bits accordingly. If and when an Executor later gives up on part of the target state, the Executor changes the corresponding abandonment bits to say so. These bits are used: (1) to answer client queries about what happened and (2) to inform the merge of target and observed state done to construct Placement Agent effective state.

The request state also includes a table that maps process role to serial number (see The Processes).

The request state also includes four sets of active process IDs, one for each kind of process (API server, placement agent, executor, observer). A process, at startup time, determines its ID (incrementing the corresponding serial number) and adds that ID to the corresponding set of active process IDs. The process monitor removes IDs from those sets.

The active placement agents are the ones that must create TSR records for a given request in order for that request to proceed to the SEMI-DONE stage (at which point those TSR records are deleted). This set is also used during failure recovery to determine which requests need to have their placement work resumed.

The set of active executor IDs is used during failure recovery to determine which VRTs need to have their execution resumed.

Observed State

The observed state is the intersection of what can be observed and what is of concern to WCC. It includes both physical and virtual resources, and relationships between them. It is at a level of abstraction that is intermediate between that of WCC requests and that of OpenStack+ APIs. It differs from WCC requests in: (a) including physical resources and their relationships with virtual ones, and (b) including virtual resources that are part of the OpenStack+ APIs but not the WCC API. Thus far the latter category is seen to consist of SDN endpoint groups, segments, and VirtNets.

We make a conceptual distinction between *WCC virtual resources* and *SDI virtual resources*. A WCC virtual resource is the sort of thing (VM, storage, or networking) that can appear in a VRT as given to WCC by a client. An SDI virtual resource is something that appears in the API of the underlying cloud: an OpenStack+ VM, storage volume, or networking thing. We expect most WCC virtual resources to eventually be implemented by SDI virtual resources. Note that SDI virtual resources have distinct IDs from even corresponding WCC virtual resources.

The observed state is sensed from the underlying cloud by the observers. We have been warned that the available sensors are not always accurate. Thus the observed state is divided into two parts: one is what the sensors have reported, and one is adjustments or "fudge factors". Some adjustments are normally determined automatically. All adjustments can be set by administrative requests. Examples of adjustments are: the amount of memory mysteriously consumed on a host, the set of hosts that are down regardless of what the sensors says (this is distinct from quiescent ones). Each adjustment decays over time. For example, an amount of memory decays exponentially;

a member of a down set evaporates a certain amount of time after it was added to that set. This decay causes WCC to eventually probe again for ground truth.

The observed state includes the current binding between WCC virtual resource ID and SDI virtual resource ID. The observed state must also effectively bind SDI virtual resources with their physical hosts. For the sake of schema consistency with the target state, the observed state includes the current binding between WCC virtual resources and their physical hosts. As a matter of de-normalizing for some convenience, the observed state could also directly record the binding of SDI virtual resources with their physical hosts.

For some kinds of virtual resources in the underlying cloud there is a way to make the underlying cloud always retain (even from the start of creation) the associated WCC ID for that resource. For example, the OpenStack command to create a VM accepts metadata that will be always associated with that VM. In such cases, the association between SDI virtual resource IDs and WCC virtual resource IDs can be discovered from the underlying cloud --- even before it has reached the WCC database.

For a sort of virtual resource in the underlying cloud (such as a VM) that, as a general rule, can be associated with a virtual resource in WCC there will occasionally be instances for which there is never such an association available. One cause of this is users going “around” WCC to create the resource directly in the underlying cloud. Another cause happens for virtual resources for which the underlying cloud does not offer to remember the associated WCC ID; if the executor that issued the create command crashes before recording the association in the target state then a successfully created virtual resource in the underlying cloud is left “orphaned”.

Target State

The target state is what the placement agents have decided should be the state of the underlying cloud, plus the request processing status information that is not part of request state. The decided state is at a level of abstraction that is intermediate between that of WCC requests and that of OpenStack+ APIs. It differs from WCC requests in: (a) including physical resources and their relationships with virtual ones, and (b) including virtual resources that are part of the OpenStack+ APIs but not the WCC API. For any VRT request whose decision has been written into the target state, an equivalent request can be reconstructed from the target state.

The decided state is divided into two parts: the target observable state (TOS) and the target non-observable state (TNOS). The target observable state has the same schema as the observed state (see exception below); the target non-observable state is the target state that has no corresponding type of information in the observed state. For example: VMs, PMs, and their binding are observable, while location constraints are not observable.

The target state includes all of the kinds of virtual resource information received in WCC requests. This includes VMs and storage volumes and their relationships. Some of this is observable and some is not.

Some of what can be observed about an underlying cloud is not under the control of WCC. Examples are capacities, grouping into a hierarchy of physical resources, and fudge factors. These are in the observed state. These are NOT in the target state.

The underlying cloud, not WCC, is in control of SDI virtual resource IDs. Thus, SDI virtual resource IDs do not appear in target state.

If the underlying cloud supports the notion of quiescence for some sort of physical resource then that is controlled through the underlying cloud and appears in WCC's observed state. WCC also supports its own notion of quiescence for some sorts of physical resource, and this notion is supported by bits in the target non-observable state.

The target state is self-consistent and consistent with the non-controllable observed state. That is, all the hard constraints are met. This is the primary part of the system where we insist on such consistency; it tames the chaos generated upstream and guides the downstream actions.

Each thing in the target state is "owned" by exactly one VRT. To enable peer-to-peer networking among some VRTs will require a new networking abstraction (SDN is going to discover this too); it does not make sense to require every peer VRT to be aware of every other peer VRT. Peer-to-peer networking among VRTs is beyond the scope of this design.

The Pub/Sub System

There is a highly available pub/sub service. It does not need to be as highly available as the database, provided that after the in-flight messages of a topic are lost we can take a WCC-specific recovery action. That action is to publish a new message, with no further details, on that topic.

The pub/sub system does not need durable subscriptions.

WCC uses three topics: C-Prod (configuration prod), A-Prod (arrival prod), and OT-Prod (observed/targeted prod). These are used only to wake up subscribers that might need to be notified of work waiting in the database. A message is published on the C-Prod topic whenever there is a change to the variable configuration of WCC itself. A message is published on the A-Prod topic whenever a new request is added into the database or otherwise set to the ARRIVED stage. A message is published to the OT-Prod topic whenever either (a) a change is written into the observed state, (b) a new placement decision has been written into the target state, or (c) an Executor has given up on part of the target state. Most of these messages carry no essential details (but might, e.g., carry the request ID to aid debugging). An OT-Prod message can optionally carry a set of pointers into target, observed, and/or request state; if there is no set of pointers then it is implicitly the union of the all the possible pointers. A pointer into observed state is either (a) the ID of a resource in that state or (b) the string "fudge factors" (which points indiscriminately at all of them). The pointer "target state" points at the target state owned by VTs affected by requests that the receiver has not yet tracked.

The Processes

The work of WCC is done primarily by four kinds of processes: API servers, placement agents, executors, and observers. For each kind of process there is a configured number to run (the place where each runs may also be configured). Each process plays a *role* in the system. A role identifies the kind of process and includes something (e.g., small integer, or place where it runs) that distinguishes one from another.

An example role might be "API server #1" or "observer on pvespa016". Each particular running process has an ID that combines its role with a serial number; successive runs of the same role use successive serial numbers. An example ID might be "placement agent #3 run 42" (or something more succinct like "P3.42").

The processes are monitored for liveness and restarted as necessary. The exact mechanism, and grouping for restart, is an open degree of freedom. However, there are requirements on performance and behavior. The process monitor knows the ID of each process it is monitoring. When the monitor decides to do a restart, the first thing it does is a database transaction that removes the ID(s) of the to-be-restarted process(es) from the corresponding set(s) of active ones and, in the case of a placement agent, removes its TSR records; if that DB transaction suffers a concurrency conflict then it is retried, repeatedly until success. Because detection of process crashes and wedges is imperfect, a process that will be restarted may yet get a few things done before its coming termination. This could potentially confuse things, by having a database transaction done by a process that the database already says is extinct. To guard against such confusion, the relevant database transactions begin by checking that the transactor's ID is still in the relevant active process ID set; if not then the process aborts the transaction and terminates itself. This is implicitly part of every such DB transaction and is not explicitly discussed in the definitions of those transactions. Specifically, these are the transactions that manipulate TSR records.

Process failure detection should be timely. For the sake of not holding up request processing due to missing TSR records we require that failure detection of placement agents be fairly prompt: only when a placement agent's ID is removed from the active set can further decision-making about involved VRTs be resumed. Removal of a terminated executor ID is not required for further decision-making, but it *is* necessary for resumption of the executing that was in progress.

Following are overviews of each kind of process. For details on their behavior see the later section on Procedures.

API Servers

An API server accepts requests from clients and takes top-level responsibility for handling those requests. For a WCC-configuration or fudge factor do-something request, the API server directly handles it; for a VRT or quiesce/un-quiesce do-something request, the API server injects that request into WCC and replies with the request's ID. For an information-seeking request, the API server gathers the requested information and returns it (arrows omitted from the picture for brevity).

Placement Agents

A placement agent does the work involved in advancing requests to the TARGETED stage. The largest part of this work is running the placement algorithm, but there are other critical steps involved.

The similar but distinct term "placement engine" refers to a body of code that makes up a large part of a placement agent but is more generic; an agent wraps its engine with the particulars of how it interacts with the other processes in this architecture. The boundary between the engine and agent is not in the scope of this document.

A placement agent handles a request by choosing a future state in which the request is satisfied; figuring out how to get from here to there is the job of the executors.

This architecture has multiple placement agents working in parallel. Since each is solving a global problem, independently from its peers, they will sometimes make conflicting decisions. We take an optimistic concurrency control approach: expecting that conflicts will be rare, the system will detect and recover from conflicts. Detection happens as the decisions are written into the target state: a decision that is inconsistent with the other database contents is not allowed to be written into the database. Recovery is by going back and reconsidering what the decision should be.

The internal structure of a placement agent includes: effective state, the placement algorithm, a set of decided requests (including their decisions), and a *finished request ID set*. That set holds the IDs of requests whose decision has been incorporated into the local effective state. The decided requests are those that the agent has run through the placement algorithm but not yet finished writing into the target state. A placement agent's *effective state* is what the agent uses when deciding what to do about requests; it is a "union" of the observed and target states (abandonment bits are ignored, all the target is used). Some changes (e.g., VM migration) take a long time to put into effect; planning against the union prevents making a new decision that cannot be put into effect until all the previous changes complete. Planning against the union also protects against nasty surprises if and when an Executor gives up on part of the target state. Ignoring the abandonment bits protects against nasty surprises if and when an abandoned change completes anyway. Each placement agent tracks the observed state and the target state updates of the others by virtue of subscribing to the OT-Prod topic and reading the relevant state after notification of an OT-Prod message. Whenever it reads the target state corresponding to a request whose stage is TARGETED, an agent adds that request's ID to the finished set. An ID is removed from the finished set once the agent notices that the identified request is in the SEMI-DONE or FULLY-DONE stage, and this does not have to be done particularly promptly (we only need to be sure it happens before the request is deleted from the on-line request state and before the agent's memory fills up with unnecessary entries in the finished set); the exact details of how the agent notices that is an available degree of freedom. Upon receipt of an OT-Prod message whose pointer set includes "target state", the relevant part of the target state is identified as that which is owned by VRTs in the VRT set of requests that are NOT in the local finished set and whose stage is TARGETED.

A placement agent's effective state tracks changes to target and observed state without attempting to preserve the ordering of those changes, and thus is not necessarily self-consistent. Rather, it is eventually consistent with the target and observed states. That is, everything written into target or observed state eventually appears in, or is overwritten (by something later) in, the effective state of each placement agent that lives long enough.

Executors

An executor works on making the true state match the target state.

The attention of the executors is focused on a set of VRTs. These VRTs are selected from those not in the DONE stage. An administrative request to quiesce a physical resource may cause several VRTs to leave the DONE stage.

Observers

The observers uses the sensing facilities of OpenStack+ to sense the true state of the underlying cloud, copy it into the observed state, and submit new requests to recover from discovered hard constraint violations. We use the term “discovery” for the special case of sensing physical resources not previously in the observed state.

Procedures

Serving API Requests

When an API server starts up it adopts an ID that is unique among all API server runs in the history of the system. The server also publishes an A-Prod message, whose purpose is to cover for the A-Prod message that this process’ predecessor possibly failed to publish. Similarly, it publishes a C-Prod message. It also publishes an OT-Prod message with the pointer “fudge factors”.

An API server checks authorization before any further handling of a request. Available freedom: how?

An API server handles a request to modify WCC configuration by first checking the request, in isolation, for whether it is reasonable. If not, an error reply is returned. Otherwise the API server does an ACID transaction that first does any more thorough request checking that is needed. If a problem is found, the transaction is ended and an error reply is returned. Otherwise, the transaction makes the requested change and then commits. If the transaction is aborted due to a concurrency conflict then the transaction is retried. After the transaction is successfully committed, a message is published to the C-Prod topic.

An API server handles a request to set a fudge factor by doing a transaction that sets the fudge factor. This request does not check hard constraints, because the fudge factor is considered a soft estimate. If the DB transaction fails due to concurrency conflicts then it is retried, with exponential back off, until success. Then the API server publishes an OT-Prod message pointing at the resource to which the fudge factor applies.

An API server handles a VRT do-something or quiesce request by (1) writing it into the request state, marked as ARRIVED, then (2) publishing an A-Prod message, then (3) replying with the request’s ID. Each do-something request is assigned an ID that is composed from the server’s ID and a serial number of requests handled by that server. The API server checks that the request is syntactically well formed, but does not do any deeper checking (such as checking for references to non-existent things). The write into the request state has two parts: (1) writing the immutable topology details (in the case of requests that have such) and (2) writing the mutable processing status. The two parts can be done in separate database transactions; indeed, the immutable topology details do not even need to be stored in a database. The two parts should be done in that order, with the mutable database contents including a pointer to the immutable stuff; in case of crash between the two parts, cleanup need not be prompt and can be guided by looking for unreferenced immutable stuff.

Placement

When a placement agent starts up it first publishes an OT-Prod message pointing to "target state", to cover for any message that its predecessor failed to publish. Then the agent subscribes to the C-Prod, A-Prod, and OT-Prod topics --- but holds off on processing notifications until all the startup procedures have finished. The agent initializes its decided requests set to empty.

The agent then does a DB transaction that:

- determines the agent's ID and adds it to the set of active placement agent IDs,
- reads the observed state, target state, and abandonment bits to initialize its effective state,
- initializes its finished request ID set to contain ID of each request whose processing stage is TARGETED,
- creates TSR records for itself paired with each request ID in the finished set and then, for each of those requests:
    - (currently the request's processing stage is TARGETED) if all the needed TSR records are present then deletes all the request's TSR records and sets its processing stage to SEMI-DONE, and then
    - if the request's processing stage is SEMI-DONE and the request has been fully executed then sets the request's processing stage to FULLY-DONE.

If this database transaction fails due to concurrency conflicts then it is retried, repeatedly with exponential back off, until success. Each try constructs the local effective state and finished request ID set from scratch. Successful completion of this transaction completes the agent's startup.

Going forward from the beginning of startup: any receipt of an OT-Prod notification causes the agent to eventually open a database transaction to update its effective state with the relevant target and/or observed state. If the message includes the pointer "target state" then the relevant target state includes that owned by any VRT that is in the VRT set of any request whose processing stage is TARGETED and whose ID is not in the local finished request ID set. For each such request, the transaction includes creating a TSR record pairing that request's ID with the agent's ID. After updating effective state and creating TSR records, the transaction considers advancing processing stages: if the request's TSR set is complete then the transaction deletes all the request's TSR records and sets the request's stage to SEMI-DONE and then, if the request has been fully executed (the request's ID is not in the targeted set of any of its VRTs), sets the request's stage to FULLY-DONE. If there is a concurrency conflict on such a transaction then it is retried, repeatedly (with exponential back-off) until success.

The agent then runs its main service loop in a single thread. The body of the service loop begins with a DB transaction that queries the request state for a workable request.

There are two kinds of workable requests: (1) a subset of those needing recovery from a previous placement agent crash and (2) a subset of those waiting to start placement work for the first time. The requests needing recovery are those for which the processing stage is PLACING and the associated placement agent ID is missing from the set of active placement agent IDs. The requests waiting to start work for the first time those in the ARRIVED stage. For both kinds, only a subset is workable. The reason for accepting only a subset is to serialize decision-making about any given VRT and serialize decision-making about any given physical resource. We say one request *intersects* another if their VRT sets intersect or both are quiesce/un-quiesce requests about the same physical resource. A waiting or recovering request R is workable if and only if it passes the *ordering criterion*. Informally, the ordering criterion for request R is that there is no intersecting earlier request that has not yet been decided. Precisely, the criterion is that there is no request S such that:

- S intersects R,
- S has an earlier submission time than R, and
- Either (a) the processing stage of S is TARGETED and there is no TSR record pairing S with the ID of the placement agent that is proposing to do the work or (b) the processing stage of S is earlier than TARGETED.

If no workable request was found then the DB transaction ends and there is a wait that ends after either (a) notification of an A-Prod message (any A-Prod that arrives after the database query started will do) or (b) creation of a TSR record involving this placement agent; the loop body is restarted after the wait.

If some workable requests were found then one is selected. Requests needing recovery take priority over others. The transaction then sets the selected request's processing stage to PLACING and sets that request's associated placement agent ID, and then commits. If the DB transaction aborts due to a concurrency conflict then it is retried with an exponential back off. Note that a retry may find no workable requests remain, in which case the earlier discussion applies. After successfully committing a transaction that finds some workable requests, the agent begins work on the selected request. First there is more checking, such as that all referenced things actually exist. For a quiesce request, as a special case, this checking includes testing whether the request's VRT set equals the set of VRTs currently using the physical resource to be quiesced. If the request's VRT set has extra VRTs then they are removed and an A-Prod message is published. If some VRTs are missing from the request's VRT set then the request is jumped back to the ARRIVED stage, with an updated VRT set, and then the agent's loop body finishes for that request (no A-Prod is needed because there is already at least one placement agent --- this one --- that will immediately look for requests on which to work). Otherwise, the regular outline continues. If any of the (other) checking finds an error in the request, the decision is to do nothing about the request except note the error in the request's success/failure state. If no error is found then the request is compared against the effective state. If this comparison reveals a need to run the placement algorithm, then it is run. That may or may not find a placement. If no placement is found then the decision is to do nothing about the request except note its failure. Otherwise we have a decision that involves doing something. Now that a decision has been made, the request ---

augmented with the decision --- is put into the agent's Decided Requests set and, if there was an error/failure, the agent does a DB transaction to update the success/failure state of the request. If that DB transaction fails due to concurrency conflicts then it is retried, with exponential back off, until success. That completes the body of the agent's main service loop.

Each placement agent also has a pool of threads that process entries in the Decided Requests set. Available freedom: how many threads? Each such thread processes one decided request at a time. The thread for a successful decided request does a series of database transactions that write the decision into the target state and update the request processing status. The last transaction updates the request processing status; it may --- and the preceding transaction certainly do --- write parts of the decision into the target state. Each transaction that updates target state also checks all the hard constraints related to the update; for a request that had already failed no constraint checking is needed. If any violation is found, the related target state update is not done (previously committed updates may remain) and the decision is retried. This involves a database transaction that sets the request's processing stage to ARRIVED, and all the other usual processing to submit a request except that the already assigned request ID is used. Otherwise (i.e., if no hard constraint violation is found) the update is committed. The transaction that updates the request processing status sets the request's processing stage to TARGETED (if successful) or FULLY-DONE (otherwise). In the success case the transaction also: (1) sets the stage of each of the request's VRTs to NEEDY, (2) adds the request's ID to the targeted request set of each VRT of the request, and (3) for each part of TOS that was updated, sets the corresponding abandonment bit to indicate the expected success in execution. The transaction then commits. A concurrency conflict is handled by retry, with exponential back off, until success. Once that transaction is committed, the thread publishes an OT-Prod message pointing at "target state" if the request was successful (and, as already implied, an A-Prod message if the request was re-submitted for retry).

Dynamic Placement Concurrency

In the above design the number of concurrent placement calculations is simply bounded by a configured number. Perhaps we can do better. Optimistic concurrency control has two competing concerns: (1) the longer one placement calculation takes, the greater the motivation for concurrency, and (2) the bigger the placement problem, the higher the probability of conflict (and suffering a conflict is worse than waiting would have been). Suppose we can develop a simple calculation that estimates the expected gain (which may be negative) from a parallel launch of a particular new placement calculation given (a) the placement problems already in concurrent progress and (b) some efficient abstract of the target and observed state. Define the *conflict criterion* for a request R to be that the expected gain of adding, to the current situation, a concurrent placement agent activation for R is positive. We can modify the definition of a workable request, by requiring that the considered request R must pass both the ordering criterion and the conflict criterion. In this way we make the degree of placement concurrency dynamic in a smart way.

Executing

Executing is the process of updating the true state to match the target state. This is organized around VRTs. There are multiple executors, and at any given time: (a) each executor is working on at most one VRT and (b) each VRT has at most one executor working on it. New decisions can be made about a VRT while a previous decision is being executed; the VRT’s executor will notice the new target state for that VRT and adjust course accordingly.

At startup, an executor subscribes to C-Prod and OT-Prod messages, but does not process notifications until it enters regular service mode. The executor then does a DB transaction that reads the relevant WCC configuration, determines the executor’s ID, adds it to the set of active executor IDs. If this DB transaction fails due to concurrency conflicts then the executor will retry, with exponential back off, until success. Then the executor enters its regular service mode.

The regular service of an executor is a loop whose body begins with a DB transaction that looks for a VRT on which to work. If this transaction fails due to concurrency conflicts then it is retried, with exponential back off, until success. The executor queries for a VRT that either (a) is in the EXECUTING stage and does not have an associated executor ID that is in the set of active executor IDs (this is a VRT for which recovery from an executor crash is needed) or (b) is in the NEEDY stage (this is the normal case). If no such VRT is found then the transaction is ended and the executor enters a wait that ends after receipt of an OT-Prod notification (any received after the start of the DB query will suffice); once the wait ends, the loop body is restarted.

In the case where some suitable VRTs are found, the transaction picks one on which to work; recoveries are preferred over the normal case. The transaction sets the stage of the chosen VRT to EXECUTING, sets its associated effector ID, and sets its success/failure state to indicate no failure. Then the executor commits the transaction.

After committing the transaction, the executor proceeds to work on the selected VRT. If this is a retry then special care is taken to discover the outcome of resource creation operations that were in progress at the time of the crash. An executor retry starts with recovering as many relevant associations between underlying cloud ID and WCC ID as possible (or, possibly, prodding the observers to do this and waiting for the result). For the sorts of resources that do not have a way to record the associated WCC ID, we accept that they will be orphaned in the case of an executor crash. When the observers discover a virtual resource that can not be associated with a WCC ID, a prod is sent to an administrator to investigate and possibly delete this now-useless virtual resource.

While working on pushing target state into true state, an executor proceeds with as much concurrency as is allowed by the hard constraints. The executor never introduces a hard constraint violation, but may find an existing violation due to an exogenous change in the observed state. Either the executor eventually succeeds, or it runs into some trouble. If the trouble may be fixed by re-trying a lower level operation, this is done. Trouble not so easily banished is preserved and surfaced. That is, the VRT’s success/failure state is set to indicate failure and provide useful details, and the relevant abandonment bits are set accordingly (see later remarks about the DB transactions for these). Every subsequent decision about that VRT will prompt another try at execution, regardless of whether there is any reason to expect better results. Possible future work: be smarter about that.

For some problems that block the executor, it can tell that re-placing would produce a new target should not have that blocking problem. For example, if the problem is an unexpected hard constraint violation involving something in the target state and something in the observed state (e.g., hypervisor crash or overload) then the executor can infer that this was due to a recent change in observed state and conclude that a new placement decision would avoid that problem. In this case, and only if automatic execution recovery is viable, the executor can (similarly to an observer discovering a hard constraint violation, see below) submit a new request for placement of that VRT. Automatic execution recovery is deemed viable if every WCC virtual resource in the VRT either: (a) is in its target state, (b) has not yet been created at all, or (c) is one for which automatic crash recovery is possible (see discussion of this in the section on Observing).

While working on pushing a VRT's target state into true state, an executor is both planning and invoking effecters to accomplish steps chosen in the plan. Occasionally the executor makes a new plan, based on the latest available observed state. Target state updates are also noted, by virtue of being prodded by OT-Prod notifications pointing to "target state" and finding that the VRT's stage has been reset to NEEDY; in this case the relevant transaction also sets the VRT's stage to EXECUTING again. The executor is required to notice target state updates and handle them before committing the transaction that indicates completion; at worst this is no more difficult than starting over.

An executor should not read all of a large VRT's target state in a single ACID transaction. But if the reading is split across multiple transactions then the executor must recognize that when the VRT's stage is reset to NEEDY this means that prior reads might have returned data that is no longer current --- in which case the executor needs to repeat those reads.

While working on pushing target state into true state, an executor may occasionally do a database transaction that updates observed state and/or abandonment bits; concurrency conflicts are handled by retry with exponential back off. The observed state updates are simply an acceleration of what the observers will eventually pick up. Like all writers of observed state, after committing the DB transaction the executor publishes an OT-Prod message pointing at the part(s) of observed state that it just updated.

Once the executor can do no more about a VRT, the executor does an ACID DB transaction that first checks the VRT's stage to see whether new target state updates have arrived; if so, the executor returns to working on that VRT's target state. Otherwise the transaction: (a) sets the VRT's stage to DONE, (b) clears the VRT's associated executor ID, (c) empties the VRT's targeted request set, (d) sets any abandonment bits not yet in the correct state, (e) updates the VRT's success/failure state if appropriate, and (f) for each request that was in the targeted set: if that request is in the SEMI-DONE stage and has been fully executed then changes the processing stage to FULLY-DONE. Concurrency conflicts are handled by retry with exponential back off.

Observing

The work of observing is partitioned among the observer processes. The partitioning is based on the hierarchy of physical machines. A certain level in the hierarchy is identified in configuration as the one that determines the partitioning. There is a partition for each node at that level. There is also a partition for network resources that are not associated with any of the other partitions.

At its startup, each observer publishes an A-Prod message, and an OT-Prod message pointing at all of the observed state, to cover for any message omitted due to the crash of its predecessor.

The partitions are divided among the observers. Each observer monitors the true state of its partitions and updates the corresponding observed state. Each DB transaction for this is retried, upon concurrency conflict, repeatedly with exponential back off until success. Then the observer publishes an OT-Prod message pointing at the parts of the observed state that were updated.

When an observer discovers a change in capacity (this includes halt/removal of a physical resource), that observer checks the hard constraints related to that capacity (including the existence of the virtual resources it was hosting). Each violation discovered is written into the post-execution violation set of the relevant VRT. Each violation is classified as either a crash or an overload. An overload is something that can be remedied by a migration (an action that does not lose application state). Each crash is either *automatically recoverable*, meaning that re-creation according to the original recipe is an effective recovery, or not, meaning that higher layers must be involved in the recovery.

For each VRT that suffers violations that are either overloads or automatically recoverable crashes, automatic recovery will follow. If there is already a request in the ARRIVED or PLACING stage having that VRT in the request's VRT set, no additional action needs to be taken. Otherwise, the observer submits a new request (following the usual outline) to set the virtual state of that VRT to what is currently recorded for that in the target state. This is for the purpose of prodding a placement agent to decide how to recover from the violation.

For each VRT that suffers violations, some of which are crashes that are not automatically recoverable, there is no automatic recovery undertaken by WCC. The existence of this state is evident in the database. WCC may take some initiative to notify the relevant client (details are an available degree of freedom).

When an observer discovers a new physical resource the observer may, depending on WCC configuration, initialize the target state for that resource to be quiescent. The arrow for this is omitted from the picture for brevity.

Fault Tolerance Consequences

The system as described above tolerates certain faults. Specifically, it tolerates failure of any number of API servers, placement agents, observers, job managers, and executors. Here we review how recovery happens for requests that have been processed to various degrees.

Consider what happens if an API server crashes while handling a request. Some requests merely request information, and the client will get an error instead of that information; the client can retry. Other requests ask for some work to be done --- and, in most cases[2], the API server's handling of that work consists entirely of a single ACID transaction on the DB. The API server may crash before, during, or after that transaction. A crash before committing that transaction leaves the system not committed to serving the request. The API server might crash between (1) committing the DB transaction and (2) publishing the corresponding prod message. The startup of the replacement API server includes publishing of prod messages that cover for any such omission. In any case, the client will get an error, but not know whether the DB transaction happened (i.e., whether or not the system is committed to doing the work requested). Most such requests are idempotent, so the client can safely retry. The one exception is a request to deploy a VRT for the first time: such a request allocates a new VRT ID and associates the given topology with that ID. A better design would separate that into two kinds of request, one to allocate a new VRT ID and one to set its topology, plus automatic garbage collection of persistently unused IDs. While that still leaves one non-idempotent kind of request (allocate new VRT ID), repeats of it cost very little.

Consider what happens if a placement agent crashes. A placement agent does three sorts of things concurrently: (1) react to OT-Prod messages, (2) make decisions, and (3) push decisions out to target state. The reaction to an OT-Prod message consists of updating internal state and writing a TSR record that will be erased when the crash is detected; no further recovery issues exist regarding this sort of work. Decision-making and writing to target state apply to requests in the PLACING stage. For a request in that stage after failure detection and restart, the placement work that was in progress has been fully or partially lost and the decision-making restarts from scratch --- but including current observed state, which may reflect partial execution of the previous decision that was partially written to target state. We accept that in this case there may be a bit of churn. For a request left in the TARGETED stage, the decision making has finished and been persisted in the target state but the OT-Prod message might not have been published; this is why placement agent startup begins with an OT-Prod message pointing at "target state".

Consider what happens if an executor crashes. It may have been working on a VRT in the EXECUTING stage. Upon restart, such VRTs are the first selected to be worked on. The executor is driven by the differences between target and observed state, and so would naturally pick up where the crashed one left off --- if there were no latency between an effecter command being issued and its being reflected in the observed state. There actually is latency, which has two effects: (1) a virtual resource in the underlying cloud whose creation was recently initiated might not yet be associated in the WCC database with the corresponding WCC-level virtual resource and (2) a state-setting command may still be in progress as recovery begins. For (1): we use available mechanisms to avoid the problem and to support discovery of those associations during recovery where possible, and accept orphans in the other cases. For (2): we suppose that state setting is idempotent and so the restarted executor can simply re-issue the relevant state-setting command.

[2] The one exception is a request to set the topology of a VRT, and its additional fault tolerance complexity is discussed in the section on how an API server handles those.

Consider what happens if an observer crashes. The crash might have happened after the observer successfully updated the observed state but before it published the corresponding OT-Prod message. This is why recovery begins with publishing an OT-Prod message that points at all of the observed state.

Available Degrees of Freedom

Following is a recapitulation of the available degrees of freedom in this design.

- How many processes (of each of the four kinds discussed here) exist and where they run.
- What sort of information (number, place, …) distinguishes one role from another for the same kind of process
- How processes are grouped for restart.
- Formulation of the details describing request and VRT failures.
- Details of authorization checking.
- How many threads a placement agent runs to write decided requests to target state.
- Source of submission time for a request.
- How a placement agent notices when finished requests have reached the SEMI-DONE or FULLY-DONE stage.
- Whether/how WCC notifies a client of various interesting milestones, such as: request fully processed, post-execution violation discovered.

Possible Future Work

Following are things that are explicitly out of scope of this design.

- Management (even mere editing) of tenants.
- Supporting peer-to-peer relationships between VRTs.
- Other kinds of underlying clouds.
- Variation in what is observable, controllable.
- Multiple underlying clouds at once.
- Concurrent decision-making about a given VRT.
- Partitioning the database.
- Partitioning the placement work.