Journal of Physics: Conference Series **513** (2014) 032043 doi:10.1088/1742-6596/513/3/032043

Request for All - A Generalized Request Framework for PhEDEx

C-H Huang[a], T Wildish[b], N Ratnikova[a], A Sanchez-Hernandez[c], X Zhang[d], N Magini[e]

[a] Fermi National Accelerator Laboratory, Batavia, Illinois, USA
[b] Princeton University, Princeton, New Jersey, USA
[c] Centro Invest. Estudios Avanz, Mexico City, Mexico
[d] Institute of High Energy Physics, Beijing, People Republic of China
[e] CERN, Geneva, Switzerland

E-mail: huangch@fnal.gov

Abstract. PhEDEx has been serving CMS community since 2004 as the data broker. Every PhEDEx operation is initiated by a request, e.g. request to move or to delete data, and so on. A request has it own life cycle, including creation, approval, notification, and book keeping and the details depend on its type. Currently, only two kinds of requests, transfer and deletion, are fully integrated in PhEDEx. They are tailored specifically to the operations' workflows. To be able to serve a new type of request it generally means a fair amount of development work.

After several many years of operation, we have gathered enough experience to rethink the request handling in PhEDEx. Generalized Request Project is set to abstract such experience and design a request system which is not tied into current workflow yet it is general enough to accommodate current and future requests.

The challenges are dealing with different stages in a request's life cycle, complexity of approval process and complexity of the ability and authority associated with each role in the context of the request.

We start with a high level abstraction driven by a deterministic finite automata, followed by a formal description and handling of approval process, followed by a set of tools that make such system friendly to the users. Since we have a formal way to describe the life of a request and a mechanism to systematically handle it, to serve a new kind of request is merely a configuration issue, adding the description of the new request rather than development effort.

In this paper, we share the design and implementation of a generalized request framework and the experience of taking an existing serving system through a re-design and re-deployment.

1. Introduction

Since its deployment in 2004, PhEDEx (Physics Experiment Data Export)[1] has been the central data management system for CMS (Compact Muon Solenoid Experiment) community. Today, it manages 60 Petabytes of data in 23 million files, in 1.6 million blocks, in 127 thousand datasets and handles all data movements among 70 sites and thousands of users all over the globe.

Every activity in PhEDEx is initiated by a request. Depending on their nature, requests may need different approval processes and different handling logics. So far PhEDEx supports two kinds of requests, transfer and deletion. They are specifically tailored to the operation's workflow and are

tightly integrated in PhEDEx. Some important business logics are implemented in the code. A request is approved or denied by a single decision made by one person with sufficient privilege.

Over the years, there are other kinds of requests that are currently handled through e-mails and human interactions. It is highly desirable to incorporate them into PhEDEx to streamline the operators workflow and provide familiar and uniform tools for managing these request types. In the current implementation, to serve a new kind of request means significant development effort. The Generalized Request project will provide a framework so that future request types can be easily implemented, including the complex approval process, in which, multiple approval parties are involved in certain logical relationship.

The concept of generalization is to abstract current processes to a conceptual level which does not depend on their essences. That is, a generic "request" that is not associated with any current "types", and a set of generic operations that deal with it. This generic request is general enough to accommodate current and future requests. Then, attach type specific information to individualize it. In the end, to create a new type of request is to "define" it to the system. It will be more an incremental configuration change rather than a code update.

In order to accomplish this, we have to think not from the procedures that need to be done for a particular request but from the stages of its general life cycle and the actions that move it from one stage to another. We define the states of a general request and it is realized by a deterministic finite state automata, DFA. The next state is a function of current state and action taking place. Therefore, the control flow, the business logic, is implied by this DFA. Not all requests go though this DFA the same way, but the mechanism is the same.

To handle a complex approval process, we need a way to represent the logical relationship among the approving parties and a way to evaluate it. We will introduce a logical expression so that complex approvals can be specified.

2. Logical Design

2.1. States of a Request

A request has eight states in its life cycle. They are *created*, *approved*, *cancelled*, *denied*, *suspended*, *locked*, *aborted* and *done*. The names carry their obvious meanings. State transition rule is defined by a deterministic finite automata, DFA, as illustrated in Fig. 1.

The terminal states are ***denied***, ***cancelled***, ***aborted*** and ***done***. Semantically, these come in pairs.

a. ***denied*** and ***cancelled*** are functionally equivalent. They apply to requests that have never been approved. However, ***denied*** means, "we refuse to do this", and ***cancelled*** means "we no longer want to do this". This distinction is important in historical records.
b. ***done*** and ***aborted*** are also clearly different. In particular, ***aborted*** means that some external action may have been taken in response to this request after it was approved. E.g, some data may have been transferred, some files may have been deleted. Moving a request to ***aborted*** does *not* mean that the external action has terminated. There may be lead-times for agents to react. In general, an ***aborted*** request will need external cleanup, either manually or by some agent depending on the type.

The different terminal states allow for cleaning out the general requests table. Old requests which have long been acted on can be detected and either archived or deleted, according to the nature of the request and the actual terminal state. It also makes for a more complete audit trail.

A request can be **locked** from the **created** state. This means someone wants to clarify something about the request, so they lock it to tell other people that there is an issue to resolve. The locking does not *technically* prevent approval, it's a *cultural* flag to tell others to pay closer attention to this request.

A request may need to be approved by several people (e.g. a **Site Admin** and a **Group Manager**). In this case, when a person approves it, the request remains in the state it was in before (either **created** or **locked**). It only transitions to **approved** when a quorum of roles have approved it. A request can

only ever become **done** if it was fully **approved** *and* the action required to complete the request was executed.

A request can be **suspended** after it has been **approved**. This may not be possible for some request types, where the transition from **approved** to **done** will be instantaneous.

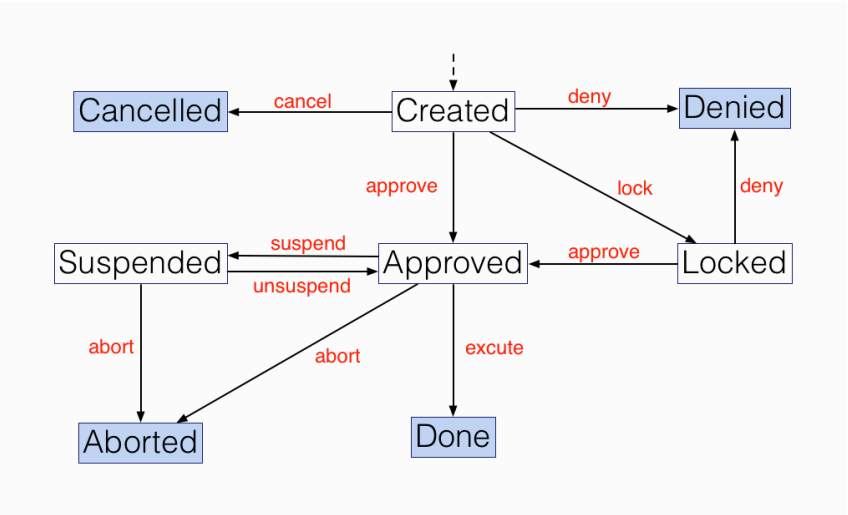


Fig 1: states of a request

An important point to note is that this DFA refers to the request itself, not to any external action that may be taken as a result of that request. In particular, many request-types will transition from **approved** to **done** automatically and immediately. E.g, a transfer request is **approved**, then a subscription is created, and the request is marked **done** immediately. The data has not yet been transferred, but as far as the *request* is concerned, the action has been taken (i.e. the subscription now exists). A corollary of this is that, to stop data flowing, you must suspend the subscription, not the request. This is, of course, identical to how things work today.

Other actions, such as creating a new database connection parameter (DBParam) file for a site, require direct operator action. The request must then be set to **done** manually. In practise, the request will be set to **done** by the tool that creates the new DBParam. This will be the general rule, the tool that performs the action will set the request state to **done** as part of that action.

2.2. Role

A "role" is an entity that acts upon the requests. In general, but not necessarily, it is a person with certain privilege. In some cases, it could be a process, acting like a person, that executes predefined actions based on predefined conditions. For example, automatically denying an unapproved request after certain period of time. A role may have many attributes, such as title and domain. However, a simple unique role definition is sufficient for the generalized request model to work. We encapsulate the complexity of role mapping in the implementation.

In reality, it is possible that more than one person may satisfy a certain role, such as the existence of multiple group managers, and that one person may assume multiple roles, such as acting as both site data manager and global admin. The exact role-to-person mapping is not a concern here. We send notifications to all possible candidates and the first one who acts fulfills the action. A person with multiple roles is assumed to act with all relevant roles at the same time.

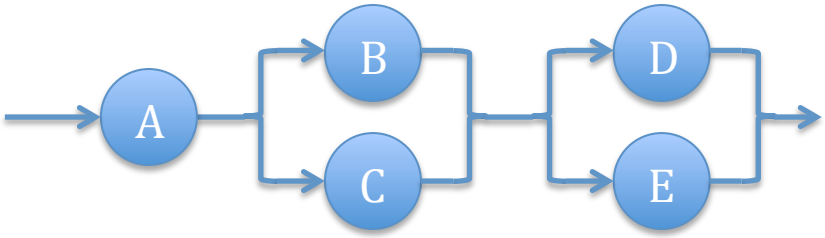
2.3. Complex Approval Process

In the DFA that represents a request's life cycle, approve and deny are simple actions. However, an approval process could be as simple as a single decision by a role or as complex as involving many

decisions from many roles, some of whom are required to make decisions and some of whom only need to do it if none of their peers made one. If we relax the dependency among the decisions, all approvals can be represented by logical expressions. The current status of an approval can be determined by evaluating its corresponding logical expression using current values of its elements.

Here are some examples:

- A: Role A decides approval or denial.
- A and B: Need approvals from both A and B. Any one denies it, it is denied.
- A or B: Either A or B can approve it. It is denied only if both A and B deny.
- A and (B or C): If A denies, it is denied. If A approves, it still needs B or C to approve it. If both B and C deny it, it is denied.
- A or (B and C): If A approves, it is approved. If A denies, it would take both B and C to approve it.
- A and (B or C) and (D or E): It can be represented in the following *Approval Plan* diagram

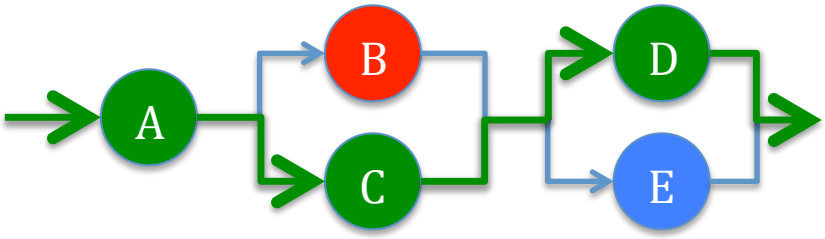


In above diagram, each bubble represents a role and has three possible values:

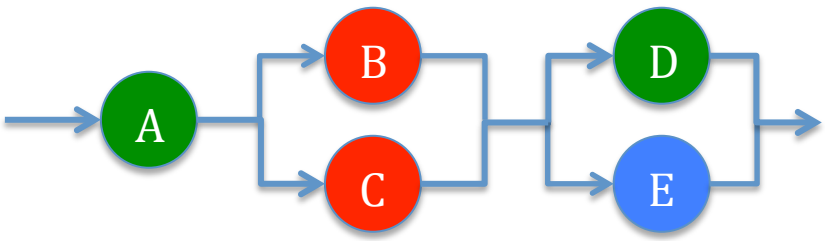
- Approved – decision is made and it is approved
- Denied – decision is made and it is denied
- Undecided – decision is not made yet and this is the initial value

Consider the diagram as a circuit. Each bubble is a switch and the lines are conductor. To conclude a request is approved is to prove a circuit exist from the beginning to the end. To conclude a request is denied is to prove no such circuit can exist. Using the last example, A and (B or C) and (D or E). The colors green, red, or blue represent an role approves, denies or is undecided.

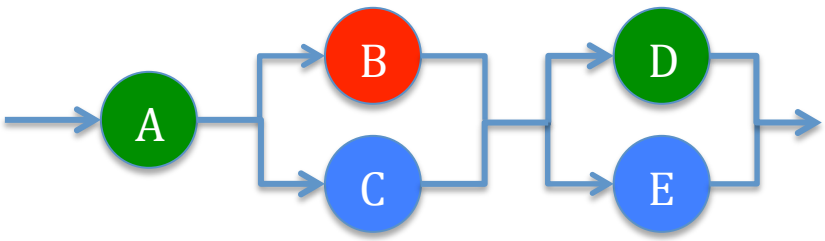
- A, C, and D approve, B denies, and E is still undecided. The request is approved since we can prove a circuit A-C-D.



- A and D approve, B and C deny, and E is undecided. The request is denied since we can prove no circuit exists.



- A and D approve, B denies, C and D are undecided. The request is undecided since we cannot either prove a circuit exist nor prove it doesn't exist.



Relaxing the dependency does not preclude dependency. A person/role making a decision can choose to delay his action upon others' decisions. Just the system would not enforce any dependency. In the end, the status of approval and denial remains the same regardless of the order of dependency.

2.3.1. Timer and default decision. A role can be given a timer and default decision. After the timer expires, if the decision is not made, the default decision will apply.

2.4. Approval plan and Approval plan template

The Approval Plan is evaluated whenever a decision is made, including the timeout and default decisions. At any given time, the evaluation returns one of the three values: *approved*, *denied*, and *undecided*. Approval plan is instantiated with supplied information at run time from a template which is predefined according to request type.

A role may appear in the approval plan only once. At instantiation, a request supplies more specific information, domain attribute, to establish the role, e.g. a specific site name for site data manager. If there is no specific information for a role, the default is taken from the template. If there are more than one for the same role, this role will be expanded to as many and all of them have to approve. For example, to establish a link between two sites, there is only one AdminSite in approval plan template. The actual request will supply two site names, say, site-A and site-B, one for each end. The instantiated approval plan will have (AdminSite(site-A) and AdminSite(site-B)). If more than one party matches a role, whoever takes the action sets the value. Once set, the value cannot be changed.

2.5. Roles and actions

Every role involved in the approval plan will be notified with a list of available actions. In addition to *approve* and *deny*, some roles will be given other action options such as *lock*, *cancel*, *suspend*, *un-suspend* and *abort*. The same option list can be retrieved any time based on the request and role.

2.6. Putting it together

- When a new request type is defined, an approval plan template is defined
- When a request is created
  - An approval plan is instantiated from the template with the specific information supplied by the request
  - From the approval plan, all parties that match the roles will be resolved and notification is sent to each one
  - The request enters created state
- Each role shall make a decision regarding that request
  - If more than one party matches a role, whoever takes the action first sets the value on behalf of that role and others cannot change it.
  - Once a decision is made, the approval plan is evaluated
  - If the evaluation results in approved, the request is moved into approved state
  - If the evaluation results in denied, the request is moved into denied state, which is final
  - Otherwise, the request remains in current state waiting for next decision to take place
- In created state, certain roles may move the request to cancelled or locked state regardless of its approval status.
- When the request reaches one of the final state, it is finished, all transactions are archived.

2.7. User interface

There are two views of this request system for the users. Both take user's role and privilege into consideration.

- Detailed request view – show everything about this request. In addition to what we have now, it will include the following:
  - Approval status - also show who has made what decision
  - Action options – based on the role and predefined privilege
- Summary view – a list of requests based on the role
  - One request on each line with essential information
  - Current state and approval status are shown
  - Based on the user's role, a list of action-options is attached to the request summary. *No action* is the default.
  - A global apply button for user to submit all selected actions

3. Applications

The following requests are to be implemented using this generalized request framework.

- Block consistency check requests
- File invalidation requests
- New database parameter requests
- Changes of SE name
- Creation of links between sites
- Unsubscribe requests
- Change of custodiality
- Re-opening a closed dataset

4. Concluding Remarks

We have addressed the design of generalized request framework and we have a prototype of some of the components, including the schema that supports it, to prove the concept. Once completed, this work will greatly streamline the work of the CMS Transfer Operations team.

References

[1] Egeland R, Metson S and Wildish T 2008 Data transfer infrastructure for CMS data taking *XII Advanced Computing and Analysis Techniques in Physics Research* (Erice, Italy: Proceedings of Science)

[2] Egeland, R, Huang, C-H and Wildish T 2010 PhEDEx Data Service. *J. Phys.: Conf. Series* **219**(062010), 2010.