draft-barton-grouptools-arch-pre03-1.html
?? Feb 2004
Tom Barton, editor
The University of Chicago
Comments to: tbarton@uchicago.edu

Group Tools Architecture

Abstract. Grouper is a project of the MACE/Internet2 Directories Working Group whose objective is to produce a set of tools that complement existing identity management systems to enable management of groups on an institutional scale. This document presents a high level architecture of that toolset, an associated data model, behavioral descriptions of the elements of the architecture, and high level specification of APIs.

1. Introduction

The MACE-Dir-Groups subgroup of the MACE/Internet2 Directories Working Group has been working towards development of a toolset to enable enterprises to manage large scale deployment of groups in conjunction with their identity management processes. The initial product of this subgroup, “Practices in Directory Groups” [1], identified several requirements for management of groups not met in commonly deployed products, which gave rise to the Grouper project (formerly called SAGE). The terminology of [1] is also followed here.

The initial step of the Grouper project was to establish a set of scenarios for use of a group toolset. This was accomplished in [2].

The present document continues the work of the Grouper project by positing an overall architecture and design for the Grouper toolset that fulfills many of the requirements of the scenarios in [2]. The design maintains a Groups Registry in which all group related information is stored. It supports

- asynchronous or batch update
- near real time provisioning
- subgroups and compound groups
- extensible typing of groups
- multiple membership lists
- automatic, configurable aging of groups and of individual memberships

Figure 1 below is a high level architectural graphic for Grouper. The behavior of each component in the architecture is described in a corresponding section of this document.

The design is currently incomplete. It is being worked on by the MACE-Dir-Groups subgroup. This document serves to approximately capture the state of that work at the present moment. When the subgroup completes this phase of the Grouper project, this document will be edited into a stable working group document expressing the final architecture and design.

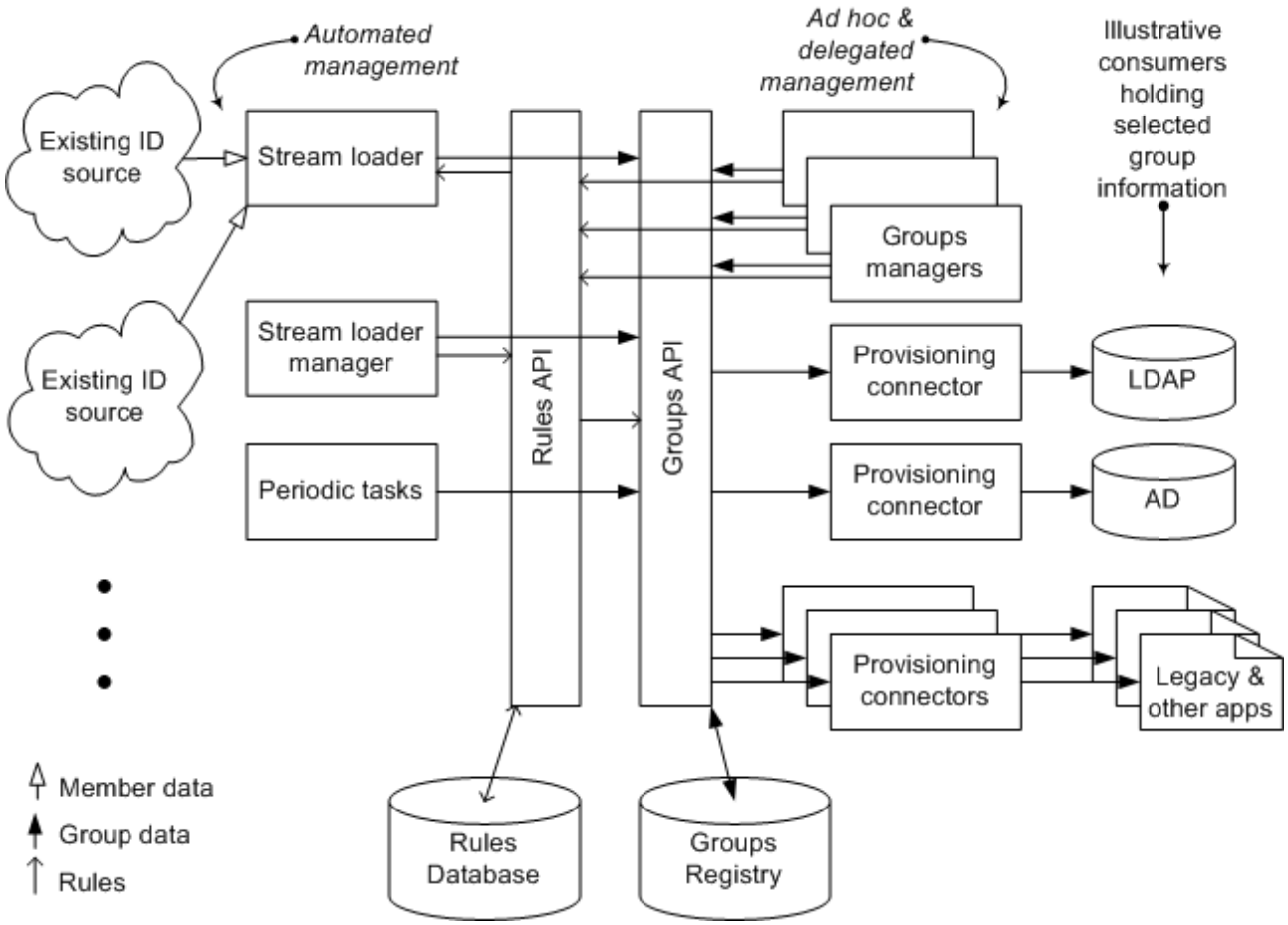

Figure 1

2. Existing ID Source

A prerequisite to implementing this architecture is a functioning identity management infrastructure. The group tools being imagined here are intended to complement and not reproduce metadirectory functionality in which data about people is integrated from a variety of systems of record. There must already be a “Person Registry”. The imagined architecture will add a “Groups Registry” to the information assets used to provision applications with the information they need.

References to member objects are treated opaquely within the Groups Registry. It is assumed that distinct memberIDs refer to distinct real world objects. Multiple sources of information about member objects can potentially be accommodated, subject to this constraint.

3. Stream Loader & its Manager

The Stream Loader reads selection and assignment rules via the Rules API and processes a stream of member records produced by one or more existing identity management systems. Selection rules enable the Stream Loader to parse member records, and assignment rules express group memberships in terms of values of parsed elements of member records. The Groups API is called to update the Groups Registry accordingly.

The Stream Loader manager is used to maintain selection logic, assignment rules, and exception lists. Different stream types can be identified and selection logic may be associated with each stream type, allowing multiple member record formats to be supported. A stream type may be marked as a “complete batch”.

Members of groups whose membership is determined by a complete batch stream's assignment rules are removed if those members fail to appear in a complete batch. Assignment rules for stream types that are not complete batches must specify a time to live for assigned memberships. A periodic task will age members out of such groups if member records conveying positive membership qualifications fail to be presented to the Stream Loader in time.

Assignment rules may refer to one or more lists of memberIDs to be exempted from membership in assigned groups. Exception lists are maintained as groups in the Groups Registry by the Stream Loader manager.

The syntax to express selection and assignment rules should be chosen so that a single (selection rule, assignment rule) pair can serve to allocate members to a set of groups. This could be accomplished by identifying assigned groups from the values of parsed elements in member records. For example, if a member record contains a memberID and a list of (department, role), an assignment rule might have the semantics of "for each (department, role), assign memberID to group named 'departmentValue-roleValue' for 2 days". The objective is to have relatively few rules result in a relatively large number of groups being maintained.

Possible stream formats to be considered for support by this architecture include flat, sql query, ldif, and specially defined xml type. Selection rule syntax may well depend upon the format. Assignment rule syntax ought to be independent of the stream format.

4. Groups Managers

In contrast to the Stream Loader, Groups Manager applications permit delegated or manual management of groups. Each Groups Manager application is tailored to manage information specific to one or more types of group (i.e., associated with a groupTypeID - see below). Examples of group types beyond a "base" type might include course groups, departmental groups, role objects, and Stream Loader exception lists. At least one Groups Manager application will be designed to enable humans to specify and modify definitions of compound groups.

We should ensure that group information managed by the Stream Loader is disjoint from group information managed by Groups Manager applications, to keep uncoordinated information sources such as programs and humans from cooking in the same kitchen. Example: the Stream Loader might populate enrollees, instructors, and basic course metadata for a course group but a Course Manager enables update of lists of TAs, course site developers, and other information not contained in the stream of identity information processed by the Stream Loader.

5. Groups Registry

Following is a description of some of the tables in the Groups Registry depicted in the entity relationship diagram below (figure 2).

The group table identifies and names each group, and also provides storage for expressions defining compound groups. Compound groups are set theoretic combinations of other groups. These are discussed further below.

Membership for all groups, and for multiple membership fields if present in the implementation, is lumped together in a single membership table. The membership table's groupFieldID field enables some groups to have more than one "membership list" associated with them, for whatever purposes (examples: RBAC; security; course groups). The memberID field can contain a groupID to support subgroups. Other references to member objects in the memberID field are treated as opaque strings. Because all membership is kept in one table, there is no membership referential integrity issue to be managed and common queries such as "list all groups to which X belongs" and "list all immediate members of group A" should be facilitated.

Each row in the member table can be marked with a time after which that row should be deleted, i.e., the member should be removed from the corresponding membership list of the corresponding group. A membership lacking a value in its removeAfter column does not expire. Expiration of memberships is a periodic task.

The metadata table stores all non-membership data for all groups.

The schema table identifies the set of types to which each group belongs. It provides a way for a group to have one or more of several sets of fields associated with it, analogous to auxiliary objectclasses in X.500 databases. For example, the "base" type inherited by all groups might include fields for the group's name, description, one membership list, and one owner list. A "course" type might add membership fields for enrollees, instructors, and TAs, as well as a course ID and other course offering metadata.

The types table lists and names the set of types that groups may have. The typeDefs tables lists the fields associated with each group type, and the fields table associates a displayable name and optional syntax with each field. These three tables would typically be read in by applications as part of their initialization prior to exercising the Groups API.

The aging table supports a periodic aging task to enable a graceful means of getting rid of stale groups.

The connectors that will be provided with changes to each group are listed in the presentation table. There is an optional indication of the type of presentation of the group in that connector. The semantics of this field are determined by the connector using it. Example: a group might have both static and forward referenced presentations maintained by an ldap connector and also be provisioned by a legacy connector that does not rely on this storage element to determine how it will present the group.

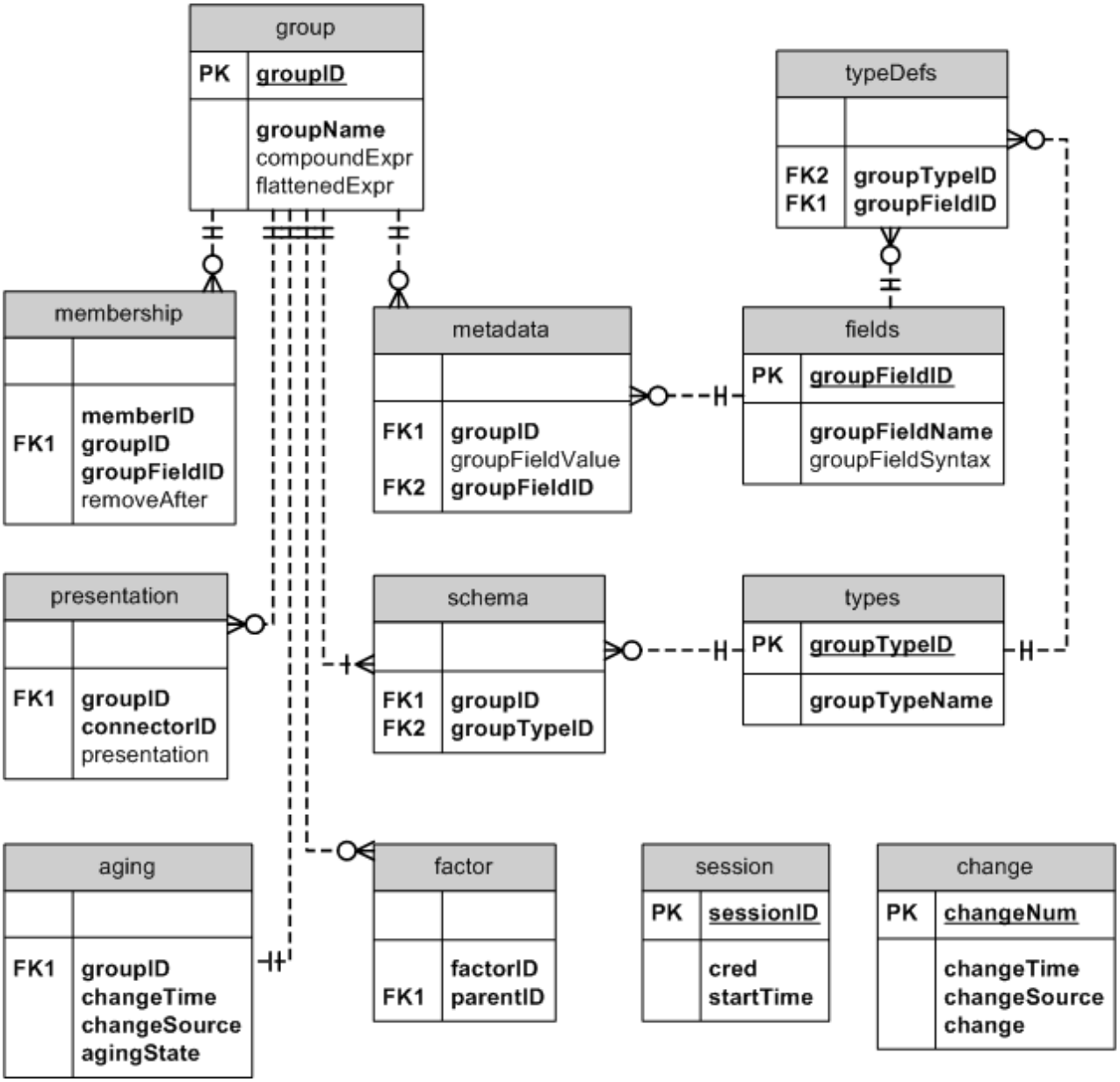

Figure 2

The change table stores a representation of each "atomic" change to the Groups Registry, as determined by the Groups API. The changeNum field is auto incrementing. The changeSource field contains the credentials of the source of the change for auditing purposes. The change field itself contains some type of representation of the change. That might be ldif-ish, xml-ish, or something else...

Compound groups are defined by a value stored in the compoundExpr field in the groups table. The syntax of this value is not yet specified, but it should express a set theoretic combination of groupIDs of other groups. The compoundExpr value is written by a Groups Manager and presumably is in a form that a human can readily manipulate.

Upon installing a new value for a group's compoundExpr, the Groups API calculates a simplified form of this defining set theoretic expression and stores the result in the flattenedExpr field. The purpose of this step is to prepare for efficient subsequent evaluation of compound group membership. Precisely how expressions are to be simplified is not yet specified, but the technique may involve repeated substitution with the goal of having the resultant flattened expression be one that can be evaluated without recursion or iteration. The factor table is provided as a means of supporting calculation of flattened expressions. As depicted here, it is represents

adjacency in the directed graph of all groups in which a directed edge from group A to group B indicates that group A appears in either the compoundExpr or the flattenedExpr defining group B.

Compound groups are considered to be of the base type only – they may not have additional types added to them. At issue is the determination of which groupFieldID should be used to express membership in a compound group. To maintain sanity, this design restricts a compound group to have only two membership list fields: owner (used by a Groups Manager to permit access to modify the compound group's definition) and members. Compound groups contribute membership to any desired list type groupFieldID of other groups by being expressed as member of the corresponding list. That is, a row of (memberID=groupID of compound group, groupID, groupFieldID) in the membership table contributes the membership of the compound group to groupID's groupFieldID membership list.

6. Groups API

The Groups API mediates all access to the Groups Registry. It integrates source information from the Stream Loader and all Groups Manager applications and mediates access to the Groups Registry by provisioning connectors and periodic tasks.

The Groups API also serializes changes from all sources to the Groups Registry and assigns a change number to each "atomic change". Change numbers are used by provisioning connectors to grab all changes of interest since a specified change number. It is anticipated that a single record presented to the Stream Loader can result in the member object it represents being added to or removed from many different groups. Hence, to preserve transactional integrity and to optimize performance of provisioning connectors, the suite of changes to membership resulting from a single changed source record ought to be considered one atomic change. Other atomic changes might be the entire specification of a new group (including its membership) or updates to the metadata of a single group resulting from a single action within a Groups Manager application.

Group creation causes issuance of a groupID and binding to a unique group name.

Illustrative API calls (note: "*" indicates a reference to a data structure):

```
List_immediate_members (groupID, groupFieldID)
List_effective_members (groupID, groupFieldID)
New_groupID (groupName)
Add_group_type (groupID, groupTypeID)
Remove_group_type (groupID, groupTypeID)
Create_group (groupID, *(groupFieldID->groupFieldValue) )
Delete_group (groupID)
Add_group_data (*(groupID->*(groupFieldID->groupFieldValue))[, TTL])
Remove_group_data (*(groupID->*(groupFieldID->groupFieldValue)) )
List_group_data (groupID)
Install_compound_expr (groupID, *(expr))
List_changes_since (changeNum, connectorID)
List_aged_groups (aTime, agingStateFilter)
Change_aging_state (groupID, agingState)
List_aged_members (aTime)
Roll_change_table (changeNum, size)
Session_start (cred)
Session_end (sessionID)
```

`Create_group`, `Delete_group`, `Add_group_data`, and `Remove_group_data` increase changeNum. All changes resulting from one call form one atomic change.

The optional TTL parameter in `Add_group_data` specifies the time to live to be applied to all membership information that is present in the data structure being passed, if any. This is used to set the value of the

removeAfter column in affected rows of the membership table. If no membership data is contained in the data structure being passed and if a TTL parameter is supplied, it is ignored.

The preceding functions are exposed to callers. Following are some envisioned to be internal to the Groups API.

```
Flatten_expr (groupID)
Update_adjacency (groupID)
Dependent_membership (groupID)
```

`Flatten_expr` and `Update_adjacency` are called by `Install_compound_expr` to update the factor table and compute the flattened form of the installed compoundExpr. Each time `Add_group_data` or `Remove_group_data` modify a row in the membership table, the groupID must be checked in the factor table to determine if the memberships of any compound groups may be impacted. `Dependent_membership` is called to perform this check and to make any dependent changes to rows in the membership table reflecting membership in each dependent compound group. Thus, all group memberships, including membership in compound groups, are always represented in the membership table.

7. Rules Database

As presently envisioned, the only role the Rules Database has in this design is to specify the operation of the Stream Loader. However, its position in the overall architecture (Figure 1) has been chosen to facilitate other, as yet undefined, uses of a more general rules database than is detailed here. Especially, discussions occurring in the MACE-Dir-Groups project at times focus on the potential desirability of incorporating support for some types of business logic into a rules database. Figure 1 leaves that prospect open for now, but as documented here the design does not require the arrow from the Rules API to the Groups API or the arrows from Groups Managers to the Rules API.

Figure 3 below models the Rules Database. The stream table lists each type of stream that is supported and flags each as being “complete” or not (cf. the Stream Loader section above). Stream types are used to enable variant sets of selector rules and assignment rules to be used on different streams presented to the Stream Loader.

The selectorRule table lists all selector expressions, the syntax of which has not yet been specified. Each selector expression is tagged with the stream type to which it pertains.

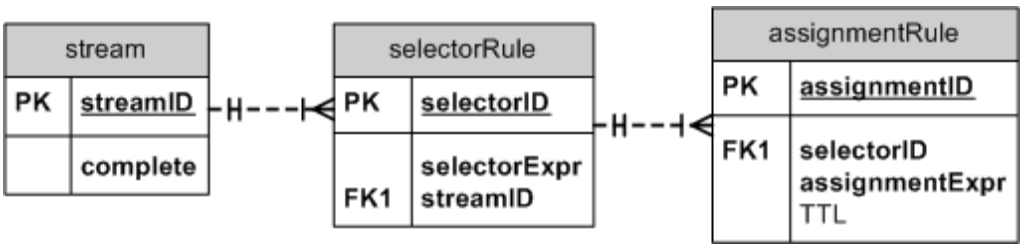

Figure 3

The assignmentRule table lists all assignment expressions. These too have yet to be specified. Each assignment rule is tied to a specific selector rule – the one that produces the tokenization of a stream record on which the assignment rule is predicated. The TTL field is used to specify time to live for memberships inferred from stream types that are not complete batches.

8. Rules API

Illustrative API calls (note: "*" indicates a reference to a data structure):

```
Create_stream (streamID, complete)
Install_selector (selectorID, streamID, *(expr))
Install_assignment (assignmentID, selectorID, *(expr)[, TTL])
Delete_stream (streamID)
Delete_selector (selectorID)
Delete_assignment (assignmentID)
Read_rules (streamID)
Read_streams ()
```

9. Provisioning Connectors

Each has a credential for accessing the Groups Registry. Each periodically requests all changes since changeNum. Saves changeNum returned by `List_changes_since` for the next polling cycle. `List_changes_since` filters information from changes by reference to the connectorID in the presentation table, so that only changes to groups being provisioned by a connector are presented to it.

Connectors are responsible for performing any meaningful referential integrity in their consumers. They are also responsible for maintaining appropriately represented groups, including spatial location in the case of an ldap directory.

Hmm, basic question: should `List_changes_since` return a table (i.e., iterate through record structures passed in RAM), an xml doc, ldif, or ... ? This is related to the question above concerning the representation of changes in the change table.

10. Periodic Tasks

To be completed…

- Aging related tasks
  1. for groups
  2. for memberships
- Trim change table
- Kill old sessions

11. Security

TBD - role structure internal to the APIs & Groups Managers.

12. References

[1] "Practices in Directory Groups", Tom Barton, Internet2 MACE-Dir working group document, October 2002. http://middleware.internet2.edu/dir/groups/internet2-mace-dir-groups-best-practices-200210.htm

[2] "SAGE Scenarios", Tom Barton, Internet2 MACE-Dir working group document, April 2003. http://middleware.internet2.edu/dir/docs/draft-internet2-mace-dir-sage-scenarios-00.html

13. Acknowledgements

This work attempts to incorporate ideas and views expressed by people participating in the MACE-Dir-Groups subgroup of the MACE/Internet2 Directories Working Group. Development of this document was

supported with funding from the University of Chicago, Internet2, and the NSF Middleware Initiative (Cooperative Agreement No. ANI-0123937).

Notes mace-dir 8 dec 2003:

- Enforce orthogonality of external memberIDs and assigned groupIDs
- Include “addAfter” in membership table so that a Groups Mgr can postdate a membership (in addition to “removeAfter”)
- Move to using UML for interface defns – think of an API as a binding of the interface to a language context.
- This presentation is too conceptually difficult – RL Bob
- Stick to groups with single membership attribute – think of a simple group rather than a “relationship object”, ie, one with multiple membership lists. – RL Bob
- What do compound groups that you don’t get with subgroups plus an exception group? – RL Bob

Craig Jurney: add “via” column to membership table for rows representing an effective rather than immediate membership. Value is groupID in which memberID is immediate member that qualified it for effective membership.

Notes mace-dir-groups 14 jan 2004:

- Change log in post release 1
- Stream loader even later than change log
- Compound groups remain in release 1
- Mulitple memberships should remain in groups registry schema, but might not be exposed in the API in release 1
- More attention to interfaces:
    - Web service?
    - Java object? (check things like perl modules that encapsulate java objects)
    - Java interface?