CCS Resource Management in Networked HPC Systems

Axel Keller and Alexander Reinefeld
PC^2 – Paderborn Center for Parallel Computing
Universität Paderborn, D-33102 Paderborn, Germany

Abstract

CCS is a resource management system for parallel high-performance computers. At the user level, CCS provides vendor-independent access to parallel systems. At the system administrator level, CCS offers tools for controlling (i.e. specifying, configuring and scheduling) the system components that are operated in a computing center. Hence the name "Computing Center Software". CCS provides:

- *hardware-independent scheduling of interactive and batch jobs,*
- *partitioning of exclusive and non-exclusive resources,*
- *open, extensible interfaces to other resource management systems,*
- *a high degree of reliability (e.g. automatic restart of crashed daemons),*
- *fault tolerance in the case of network breakdowns.*

In this paper, we describe CCS as one important component for the access, job distribution, and administration of networked HPC systems in a metacomputing environment.

1 Introduction

With the increasing availability of fast interconnection networks high-performance computing has undergone a metamorphosis from the use of local computing facilities towards a distributed, network-centered computing paradigm. The motivation is to better utilize the available hardware by linking LAN/WAN connected supercomputers to a virtual metacomputer [33]. While at the time being, there are only few multi-site applications that fully exploit the computational power of distributed nodes, metacomputing is already used on a broader scale for job load sharing and fault tolerance purposes.

Distributed high-performance computing environments usually comprise a wide spectrum of resources with different capabilities. Here, a resource management system must be able to cope with unreliable networks and with heterogeneity at multiple levels (e.g. administrative domains, scheduling policies, operating systems, protocols etc.). Because of the dynamic nature of the metacomputing components, the available resources should be identified at runtime. As an example, the performance of shared media (networks) varies over time, requiring constant updates on the global system state and structure.

From the user's point of view, a metacomputer should be as easy to use as the workstation on his/her desk. This means that users need a vendor-independent access interface and their applications should be transparently mapped onto a set of suitable target platforms.

In this paper, we present the architecture of our *CCS Computing Center Software*. Having started in 1992 with a comprehensive system that manages all computers in a site but gives users only access to a single system at a time, CCS was continuously upgraded. It now supports multi-site applications and has an open interface to metacomputer management services. As a long-term goal we want to integrate CCS—among other resource management systems—into an open global metacomputing environment.

In the 4^{th} release, three new concepts have been introduced: *'CCS Islands'* provide management facilities for administrating single HPC systems in a local site. At the next higher level, a *'Center Resource Manager'* coordinates the cooperative administration and use of all systems in a computing center, whereas the *'Center Information Server'* provides an active directory service for metacomputer access from the outside world.

All modules build on the generic *'Resource and Service Description'* that is used for specifying hardware and software components.

The contents of this paper are as follows: First, we put CCS in the context of the Globus project. Then we present the architecture of the CCS islands and their technical implementation. Thereafter, we introduce the concepts required for metacomputing (Center Resource Manager and Center Information Service) and how they work together. Our tools for re-

source and service description are described in Section 5 and some preliminary results on the use of CCS in an industrial metacomputing setting are discussed in Section 6. Section 7 gives a brief review on related projects and Section 8 presents a summary.

2 CCS – A Link to Globus?

While CCS may be seen as "just another resource management software" we have always put it in a much broader context. In fact, our primary design goal was to provide a resource management system that can be integrated into metacomputer environments like our *Metacomputer Online* toolbox [32].

Because *Globus* [15], as part of the National Computational Science Alliance [34], is certainly the most well-known metacomputing project throughout the world, we now put CCS in relation to Globus. The following list gives the most important similarities and differences between the two projects.

The Globus project regards a metacomputer as a networked virtual supercomputer constructed dynamically from geographically distributed resources that are linked by high-speed networks. It aims at a vertically integrated treatment of application, middleware and network and it provides a basic infrastructure of tools building on each other:

- resource (al)location:
  Globus Resource Manager GRM [23]
- communication layer:
  Nexus [13]
- unified resource information service:
  Metacomputing Directory Service MDS [12]
- authentication interface:
  Generic Security System GSS [26]
- data access:
  Remote IO Facility RIO [16].

In the Globus metacomputer, applications are expected to configure themselves to fit the execution environment delivered by the metacomputing system, and then adapt their behavior to subsequent changes in the resource characteristics. This concept has been named 'Adaptive Wide Area Resource Environment' AWARE.

The Computing Center Software CCS was primarily designed to manage the resources in a *single* site. They may be geographically distributed but operate in a single NFS/NIS domain. It provides an open interface so that several sites may be joined by higher-level tools—a modular approach that proved useful in several industrial projects. CCS has a hierarchical structure with autonomous software layers that interact only via message passing: The lowest level is a self-sufficient 'island' controlling a single machine or cluster which can be operated stand-alone. The next higher level consists of the Center Resource Manager (CRM) and the Center-Information Server (CIS) which build the interfaces of a site to the 'outside world'.

CCS does not only provide a comfortable user interface, but it also offers a versatile, almost system-independent interface for the administrator. Its open framework architecture allows to integrate all kinds of HPC systems. Compared to Globus, CCS

- does not support metacomputing by itself, but it provides one important component,
- has not yet an API,
- does not support remote I/O,
- has no dedicated authentication interface.

3 Computing Center Software

When the CCS project [9, 30, 31] started in 1992, only few competitive systems were available. With our background in the operation of *massively* parallel computing systems, we aimed at providing

- concurrent user access to exclusively owned resources
- interactive and batch processing at the same time,
- optimal system utilization by dynamical partitioning and scheduling,
- maximum fault tolerance for remote access via WANs.

CCS became first operationable on a 1024-node transputer system and was later adapted to PowerPlus systems from Parsytec and also to workstation clusters. Aiming at portability, we designed CCS to run on UNIX systems, such as Linux, SunOS, Solaris, AIX and others. The architecture with its modular frame structure allows to integrate a great variety of other systems, cf. Sec. 3.2.

3.1 Architecture

Island Concept. With its distributed nature, fault tolerance is a basic prerequisite for CCS working correctly. In earlier versions, the machines of a computing center were managed by a single (but physically distributed) CCS software. This caused bottlenecks

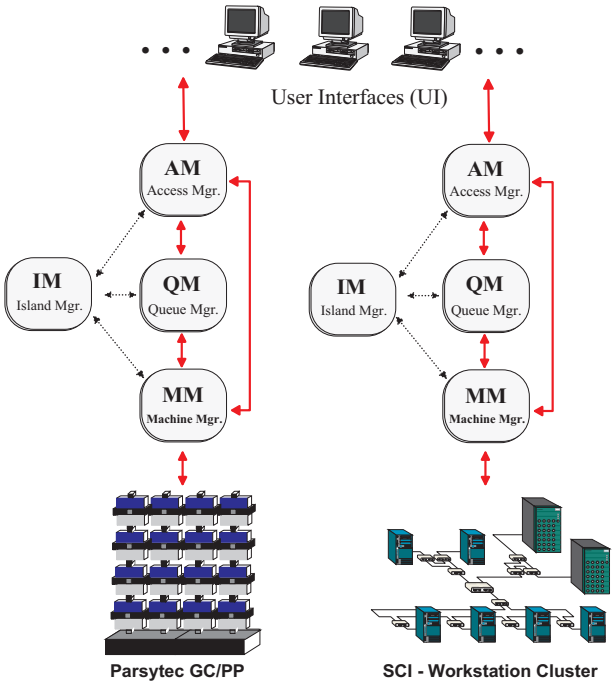

Figure 1: Architecture of CCS 'islands'

at the single scheduler serving all machines and resulted in poor fault tolerance due to the one central request handler. With the 4^{th} release, each machine is now managed by a dedicated CCS, resulting in set of comprehensive, self-sufficient *'islands'* shown in Figure 1. Each island has six components, which will be described in more detail later in the text:

- The *User Interface (UI)* offers X- or ASCII-access to all capabilities of a machine. It encapsulates the physical and technical characteristics for a homogeneous access to single or multiple heterogeneous systems.
- The *Access Manager (AM)* manages the user interfaces and is responsible for authorization and accounting.
- The *Queue Manager (QM)* schedules the user requests.
- The *Machine Manager (MM)* manages the parallel system.
- The *Island Manager (IM)* provides name services and watchdog functionalities for reliability.
- The *Operator Shell (OS)*, not shown in Figure 1, allows system administrators to control CCS, e.g. by connecting to the single daemons.

With the island concept scalability, reliability and error recovery have been improved by separating the management of different machines into different islands. Each machine uses a dedicated scheduling strategy and can therefore be operated in a different mode (batch, shared, mixed etc.). Specific user interfaces can be used to reflect special system features.

Reliability. In heterogeneous distributed environments, reliability is of prime importance. For example, a message-passing program that does not receive an answer from its partner in time, does not know

- whether the network is down,
- or whether it temporarily has a low bandwidth,
- or whether the communication partner has died.

This is because the necessary information is not available at OSI level 7. We therefore need an instance with global and up-to-date information on the status of all system components. This instance should be always accessible and it should have little or no dependencies on other modules.

In CCS, this instance is the *Island Manager (IM)*. At startup and shutdown time all CCS daemons notify the IM. Hence the IM has a consistent view on the current status of the processes in an island. The IM is authorized to stop erroneous daemons or to restart crashed ones.

In its second task, the IM provides name services. It maintains an address translation table that matches symbolic names to the daemons' physical network address (host ID and port number). This gives a level of indirection, allowing the IM to migrate daemons to other hosts in the case of overloads or system crashes. Symbolic names are given by the triple `<center, island, process>`. As a side effect, this allows to run several CCS islands on a single host concurrently.

User Management. The *User Interface (UI)* runs in a standard UNIX shell environment like tcsh, bsh, or ssh. Common UNIX mechanisms for IO-redirection, piping and shell scripts can be used and all job control signals (ctl-z, ctl-c, ...) are supported. Five CCS commands are available:

- *ccsalloc* for allocating and/or reserving resources,
- *ccsbind* for re-connecting to a lost interactive application/session,
- *ccsinfo* for displaying information on the job schedule, users, job status etc.,
- *ccsrun* for starting jobs on previously reserved resources,
- *ccskill* for resetting or killing jobs and/or for releasing resources.

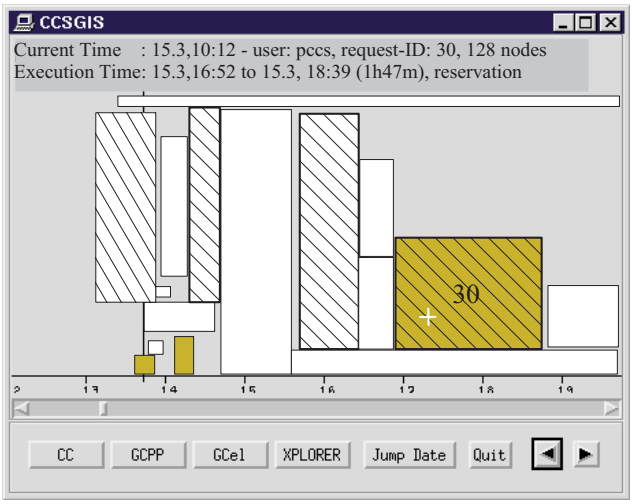

Figure 2: Scheduler GUI displaying the scheduled nodes (vertical axis) over the time axis.

The *Access Manager (AM)* analyzes user requests and is responsible for authentication, authorization and accounting.

CCS supports project specific user management. Privileges can be granted to either a whole project or to specific project members, for example

- access rights per machine
- allowed time of usage (day, night, weekend, ...)
- maximum number of concurrently used resources
- accounting per machine (product of CPU-time and #PEs)
- machine access rights (batch, interactive, the right for reserving resources)

User requests are sent to the *Queue Manager (QM)* which schedules the jobs according to the current scheduling policy. CCS provides several scheduling modules (FCFS, FFIH, FFDH, IVS) that can be plugged in by the system administrator, cf. Sec. 3.2 [18].

Job Scheduling. The first CCS release was capable of managing exclusive (non-timeshared) resources only. With release 4.0, CCS has been upgraded to support time-shared resources as well. As in Condor, Codine, or LSF, the system administrator may specify a maximum load factor that is allowed on the single nodes.

In their resource requests, users must also specify the expected finishing time of their jobs. Based on this information, CCS determines a fair and deterministic schedule. Both, batch and interactive requests are processed in the same scheduler queue. The request scheduling problem is modeled as an n-dimensional bin packing problem, where the one dimension corresponds to the continuous time flow, and the other $n-1$ dimensions represent system characteristics, such as the number of processor elements. Currently, CCS uses an enhanced first-come-first-serve (FCFS) scheduler, which fits best to the request profile in our center. The waiting times are reduced by first checking whether a newly incoming request may fit into a gap of the current schedule. The current schedule is displayed in an X-window as illustrated in Figure 2.

CCS allows to reserve resources for a given time in the future. This is a convenient feature when planning interactive sessions or online events. As an example, consider a user wants to run a parallel application with 64 processors of the Parsytec GCel from 9 to 11 am at 13.2.1999. This resource allocation is done with the command:
`ccsalloc -m GCel -p 64 -s 9:13.2.99 -t 2h`.

'Deadline scheduling' is another useful feature. Here, CCS guarantees the job to be completed at (or before) the specified time. A typical scenario for this feature is an overnight run that must be finished when the user comes back to his/her office in the next morning. Deadline scheduling gives CCS the flexibility to improve the system utilization by scheduling batch jobs at the earliest convenient and at the latest possible time.

The CCS scheduler is able to a handle two kinds of requests, those that are fixed in time and the variable ones. A resource that has been reserved for a given time frame is fixed: It cannot be shifted on the time axis (see the hatched rectangles in Fig. 2). Interactive requests, in contrast, can be scheduled earlier but not later than asked for. Such shifts on the time axis might occur when resources are released before their estimated finishing time.

System Partitioning. For metacomputing, we need a scheduler that computes deterministic schedules. Additional design objectives were optimal system utilization combined with a high degree of system independence. To deal with these conflicting requirements we have split the scheduler software into two parts, one of them (QM) being completely independent of the underlying hardware architecture. With this separation, the scheduler daemon has no information on the mapping constraints such as the minimum cluster size, or the amount/location of the link entries.

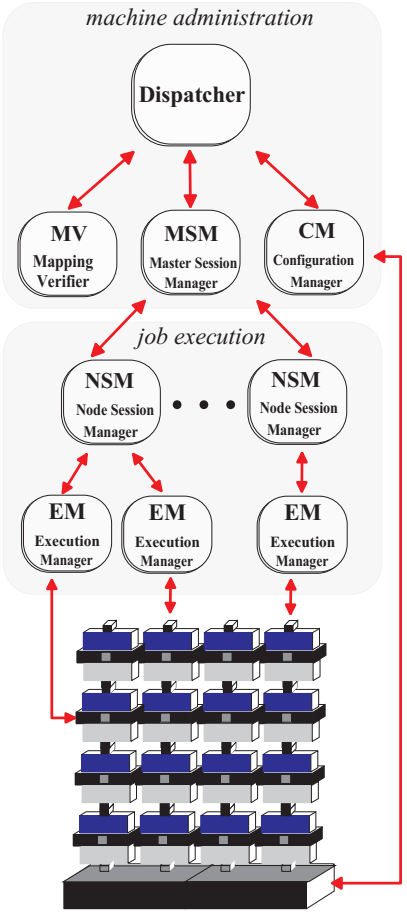

Figure 3: Detailed view of the machine manager (MM)

These machine dependent tasks are performed by a separate instance, the *Machine Manager (MM)*. The MM verifies whether a schedule given by the QM can be mapped onto the hardware at the specified time, now also taking concurrent use by other applications into account. If the schedule cannot be mapped onto the machine, the MM returns an alternative schedule to the QM.

The separation between the hardware-independent QM and the system-specific MM also allows to employ system-dependent mapping heuristics that are implemented in small system-specific modules. Special requests for IO-nodes, partition shapes, memory constraints, etc. are taken into consideration in the verifying process. Moreover, with the machine-specific information encapsulated in the MM, CCS islands can be easily adapted to other architectures.

Process Creation and Control. At configuration time, the QM sends the user request to the MM. The MM then allocates the compute nodes, loads and starts the application code and releases the resources after the run.

Because the MM also verifies the schedule, which is a polynomial or NP-hard problem, a single MM daemon might become a computational bottleneck. We have therefore split the MM into two parts, one for the machine administration and one for the job execution (see Figure 3). Each part contains a number of modules and/or daemons.

The machine administration part consists of three separate daemons (MV, MSM, CM) that execute asynchronously as shown in Figure 3. A small *Dispatcher* coordinates the lower-level components.

The *Machine Verifier (MV)* checks whether the schedule given by the QM can be realized at the specified time with the specified resources. Based on its more detailed information on the machine structure (hardware and software) it runs system-specific scheduling and partitioning schemes. The resulting schedule is then returned to the QM.

The *Configuration Manager (CM)* provides the interface to the hardware. It is responsible for booting, partitioning, and shutting down the operating system software. Depending on the system's capabilities, the CM may gather subsequent requests and re-organize or combine them for improving the throughput—analogously to a hard disk controller.

The *Master Session Manager (MSM)* interfaces to the job execution level. It sets up the session, including application-specific pre- or post-processing, and it maintains information on the status of the application.

It allocates and synchronizes the system entries of the user partition with the help of the *Node Session Manager (NSM)*, that is run on each specified entry node. The NSM starts and stops jobs and it controls the processes. When receiving a command from the MSM, the NSM starts an *Execution Manager (EM)* which establishes the user environment (UID, shell settings, environment variables, etc.) and starts the user application.

On time-sharing systems, the NSM invokes as many EMs as needed. It also gathers dynamic load data and sends it to the MM and QM where it is used for scheduling and mapping purposes.

Virtual Terminal Concept. With the increasing use of supercomputers for *interactive* simulation and design, the support of remote access via WANs becomes more and more important. Unpredictable behavior and even temporary breakdowns of the network should (ideally) be hidden from the user.

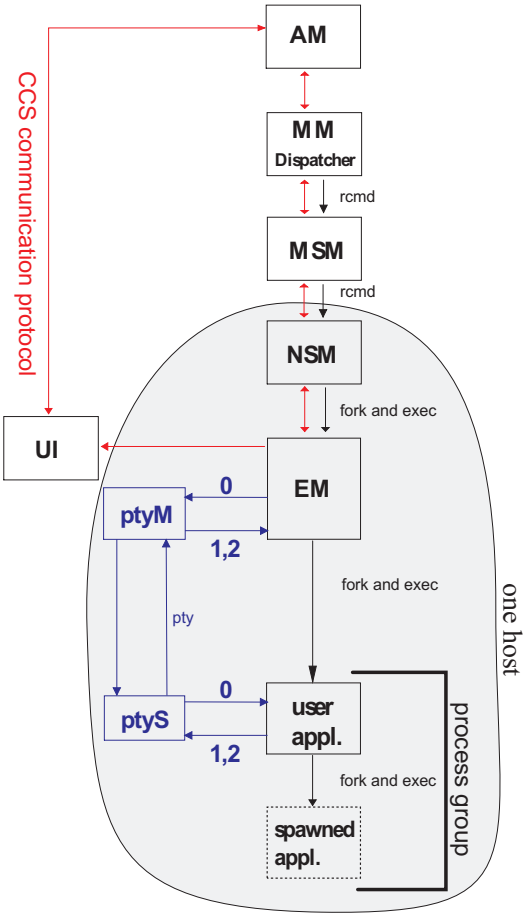

Figure 4: Control and data flow in CCS

In CCS, this is done by the EM which buffers the standard IO streams (stdin, stdout, stderr) of the user application. In case of a network break down, all open output streams are sent by e-mail to the user or they are written into a file when specified by the user. Users can re-bind to interrupted sessions, provided that the application is still running. CCS guarantees that no data is lost in the meantime.

In summary, Figure 4 gives a overview on the control and data flow in a CCS island.

3.2 Implementation Aspects

CCS consists of about 180,000 lines of C code. The code is—as far as possible—ANSI compliant and POSIX 1003.1-1990 conform. It follows a 'programming frame approach' by splitting most of the modules into two parts, a generic and a system-specific one.

As an example, Figure 5 shows the MM frame. Here, only the mapping module is machine dependent, all other parts are generic and can be re-used.

The daemons are driven by events from incoming

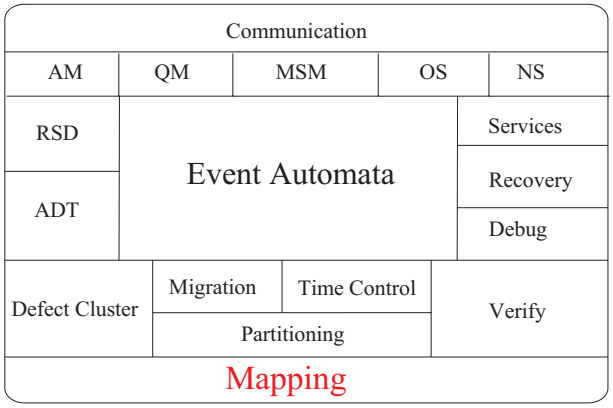

Figure 5: The MM frame

requests or timeouts. Each daemon has a watchdog which checks whether the daemon is still alive. If not, the watchdog shuts its daemon down. This is recognized by the IM, which in turn restarts the dead daemon and informs its communication partners to re-bind to the new instance.

All daemons include a wrapped timer that creates clock ticks for debugging purposes and for simulating incoming requests on a variable time scale.

Even though CCS is POSIX-conform, we implemented a *Runtime Environment (RTE)* layer that wraps system calls. This allows for easy porting to new operating systems. Currently, the RTE provides interfaces for

- the management of dynamic memory, including debugging and usage logging,
- signal handling,
- file I/O including filter routines for ASCII-files,
- manipulation of the process environment,
- terminal handling (e.g. pty),
- sending e-mails,
- logging of warnings and error messages.

The integration of new schedulers is easy, because the QM has an API to plug in new modules. This also allows the QM to use several schedulers. At runtime, the QM takes the decision which scheduler to use, thereby adjusting to specific operating modes (e.g. interactive use only or mixed time sharing and space sharing).

Communication Layer. The communication layer separates a daemon's code from the communication network, allowing to change communication protocols without the need to change the source code of the

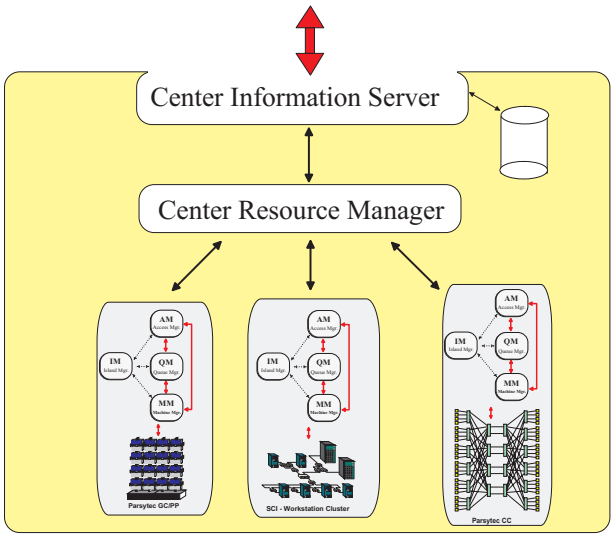

Figure 6: A site managed by CCS

daemons. The communication layer performs the following tasks:

- it provides a reliable and hardware-independent exchange of data,
- it allows to dynamically connect/disconnect to communication partners,
- it checks the availability of communication partners (in cooperation with the IM),
- it translates symbolic module names (in cooperation with the IM).

In our current implementation the daemons communicate through remote procedure calls (RPC). Compared to the faster TCP/IP sockets, asynchronous RPCs provide a more high-level method for process interaction—closely related to the client-server model of distributed computing. Also, they support data conversion with the XDR-library.

The binding of incoming RPC calls (events) to the corresponding callback-functions is done during runtime to allow to dynamically add new events by registering the corresponding event handler.

4 CCS Interface to Metacomputing

With the autonomous islands described in the last section we have one important component for metacomputer management. The three other components are:

- A passive instance that maintains up-to-date information on the system structure and state.
- An active instance that is responsible for the location and allocation of resources within a center. It also coordinates the concurrent use of several systems, which may be administered by different local resource management systems.
- A powerful but user-friendly tool that allows system administrators and users to specify classes of resources.

These three components are: The center information manager CIS, the center resource manager CRM, and the resource and services description RSD.

Center Information Server (CIS). The CIS is the 'big brother' of the island manager (IM) at the next higher level, the metacomputer level. Like the UNIX Network Information Service NIS or the Globus Metacomputing Directory Service MDS, our CIS provides up-to-date information on the resources in a site. Compared to the active IM in the islands, CIS is a passive component.

At startup time, or when the configuration has been changed, an island signs on at the CIS and informs it about the topology of its machines, the available system software, the features of the user-interfaces, the communication interfaces and so on. The CIS maintains a database on the network protocols, the system software (programming models, libraries, etc.) and the time constraints (for specific connections, etc.). The CIS also plays the role of a 'docking station' for mobile-agent software or external users.

For the higher level metacomputer components, the CIS data must be compatible or easily convertible to the formats used by other resource management systems.

The Center Resource Manager (CRM). Like the Globus resource manager, the CRM is a high-level but independent tool that lies on top of the CCS islands. It supports the set-up and execution of multi-site applications running concurrently on several platforms. The term multi-site application can be understood in two ways: It could be just one application that runs on several machines without explicitly being programmed for that execution mode [17], or it could comprise different modules, each of them executing on a machine that is best suited for running that specific piece of code. In the latter case the modules can be implemented in different programming languages using different message passing libraries (e.g. PVM, MPI, PARIX, MPL etc.). Multi-communication tools like

PLUS [6] are necessary to make this kind of multiple-site application possible.

For executing multi-site applications three tasks need to be done:

- locating the resources,
- allocating the resources,
- starting and terminating the modules.

For locating the resources, the CRM maps the user request (given in RSD notation) against the static and dynamic information on the available system resources.

The static information (e.g. topology of a single machine or the site) has been specified by the system administrator, while the dynamic information (e.g. state of an individual machine, network characteristics etc.) is gathered at runtime. All this information is provided by the CIS. Since our resource description language is able to describe dependency graphs, a user may additionally specify the required communication bandwidth for his/her application. In the mapping and migration process, the communication pattern should also be taken into account. Data on the previous runtime behavior can be gathered and condensed in an execution profile as described in [19].

After the target resources have been located, they must be allocated. This can be done in analogy to the two-phase-commit protocol in distributed database management systems: The CRM requests the allocation of all required resources at all involved islands. If not all resources were available, it either reschedules the job or it denies the user request. Otherwise the job can now be started in a synchronized way. Here, machine-specific preprocessing tasks or inter-machine specific initializations (e.g. starting of special daemons) must be initialized.

Analogously to the islands level, the CRM is able to migrate user resources between machines to achieve a better utilization. Accounting and authorization at the metacomputer level can also be negotiated at this layer.

The CRM can be implemented in several ways. As an example, it could be implemented as a single daemon or in the form of distributed instances like the QM-MM complex at the islands level.

5 Resource and Service Description

CCS includes a versatile resource description facility, named *RSD* for *Resource and Service Description*. RSD is used

- at the administrator level for describing type and topology of the available resources, and

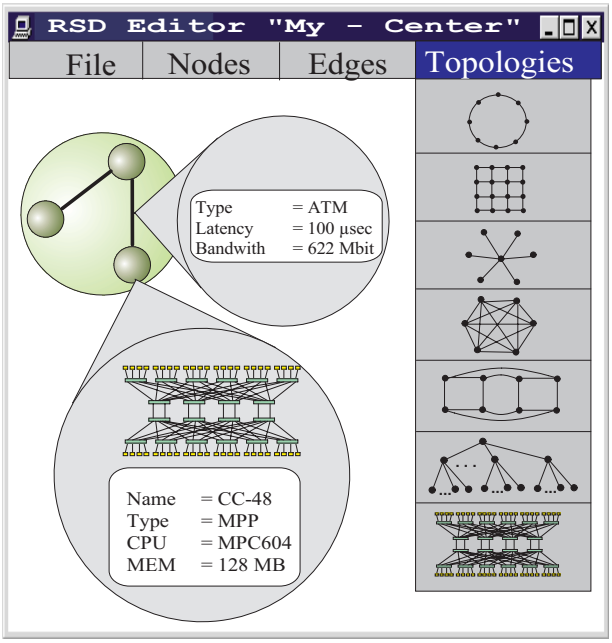

Figure 7: Graphical RSD editor

- at the user level for specifying the required system configuration for a given application.

The predecessor of RSD, the resource description language RDL [3] served in earlier releases of CCS. In general, it was regarded as too complex. Especially industrial users did not want to take the burden of typing in a textual resource specification when just wanting to run a code on a machine with a simple, regular topology. It seems, that it was too early for the user community to appreciate the full descriptive power of a versatile description language. Hence, we hided the language interface by easy-to-use command line options. But of course, RDL was still used behind.

With the current trend to distributed computing, resource description tools become important again. Based on our experiences with RDL, we now provide a more generic approach with three interfaces:

- a graphical interface (GUI) for specifying simple topologies and attributes,
- a language interface for specifying more complex and repetitive graphs (mainly intended for the system administrator), and
- an application programming interface (API) for access from within an application program.

The graphical editor stores the graphical and textual data in an internal data representation. This data

is bundled with the API access methods and sent as an attributed object to the target systems, where it is matched against other hard- or software descriptions.

The internal data description can only be accessed through the API. For later modifications it is retranslated into its original form of graphic primitives and textual components. This is possible, because the internal data representation also contains a description of the component's graphical layout. In the following, we describe the core components of RSD in more detail.

Graphical Interface. The graphical editor provides a set of simple modules that can be edited and linked together to build a dependency graph of the requested resources or a system description.

At the *administrator level*, the GUI is used to describe a center's resource components in a top down manner, starting at the outermost interconnection topology, see Figure 7. With drag and drop techniques, the administrator specifies the available machines, their links and the interconnection to the outside world.

In the next step, the machines are specified in more detail by clicking on a node. The editor then opens a window to display detailed information on the machine, if available. The GUI offers a set of standard machine layouts and some generic topologies like tree, grid or hypercube. The size and shape is defined according to the available hardware. For a single node, detailed attributes like network interface cards, disk sizes, I/O throughput, or the automatic start of daemons may be specified.

Language Interface. From a system administrator's point of view, graphical user interfaces are not powerful enough for describing complex metacomputing environments with a large number of services and resources. Administrators need an additional tool for specifying irregularly interconnected, attributed structures.

Hence, we devised a language interface that is used to specify arbitrary topologies. The hierarchical concept allows different dependency graphs to be grouped for building even more complex nodes, that is hypernodes. For a complete formal definition of the language interface see [7].

Figures 9 and 10 illustrate a resource specification for a metacomputer application `Meta` running on two systems as shown in Figure 8. The metacomputer comprises an SCI workstation cluster and

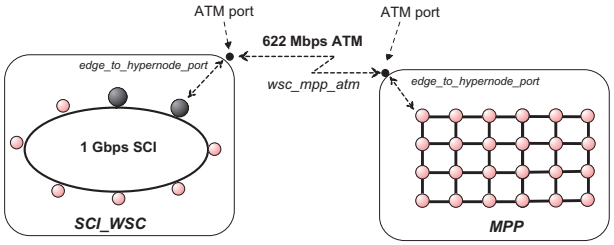

Figure 8: RSD example for multi-site application

```
NODE Meta {
  DEFINITION:
     PORT[] = (SCI, ATM, FDDI);   -- multi-valued attribute

  DECLARATION:           -- include the two hyper nodes
     INCLUDE "SCI_WSC";
     INCLUDE "MPP";

  CONNECTION:- of the MPP with SCI workstation cluster
        EDGE wsc_mpp_atm {
        NODE SCI_WSC PORT ATM <=> NODE MPP PORT ATM;};
                     BANDWIDTH = 622 Mbps;};
};
```

Figure 9: RSD specification of Fig. 8

a massively parallel system, interconnected by a bidirectional ATM network.

The definition of `Meta` is straight-forward, see Figure 9. Figure 10 shows the specification of the SCI cluster component, consisting of 8 nodes, two of them with quad-processor systems. For each node, the following attributes are specified: CPU type, memory per node, operating system, and the port of the SCI link. All nodes are interconnected by a uni-directional SCI ring with 1.0 Gbps. In the example, the first node is the gateway to the workstation cluster. It presents its ATM port to the next higher node level (see ASSIGN statement in Fig. 10) to allow for remote connections.

Internal Data Representation. The abstract data type establishes the link between the graphical and the text based representation. It is also used to store descriptions on disk and to exchange them across networks. The internal data representation must be capable of describing the following properties:

- arbitrary graph structures
- hierarchical systems or organizations
- nodes and edges with arbitrary sets of valued attributes

```
NODE SCI_WSC {
  DEFINITION:
  CONST N = 8;        -- number of nodes
  SHARED;             -- allocate resources for shared use

  DECLARATION:
  -- we have 2 SMP nodes (gateways), each with 4 processors
  -- each gateway provides one SCI and one ATM port
  FOR i=0 TO 1 DO
     NODE i {
     DECLARATION:
     CPU=ALPHA; MEMORY=512; MULTI_PROC=4; PORT=[SCI,ATM];
     };
  OD

  -- the others are single processor nodes
  -- each with one SCI port
  FOR i=2 TO N-1 DO
     NODE i {
     DECLARATION:
     CPU=ALPHA; MEMORY=256; OS=SOLARIS;PORT=SCI;
     };
  OD

  CONNECTION:
  -- build the 1.0 Gbps unidirectional ring
  FOR i=0 TO N-1 DO
     EDGE edge_$i_to_$((i+1) MOD N)
     { NODE i PORT SCI => NODE ((i+1) MOD N) PORT SCI;
     BANDWIDTH = 1.0 Gbps;
     };
  OD

  -- establish a special virtual edge from node 0 to the
  -- port of the hyper node SCI_WSC (=outside world)
  ASSIGN edge_to_hypernode_port
  { NODE 0 PORT ATM <=> PORT ATM;};
};
```

Figure 10: RSD specification of the SCI part

Furthermore it should be possible to reconstruct the original representation, either graphical or text based. This facilitates the maintenance of large descriptions (e.g. a complex HPC center) and allows visualization at remote sites.

In order to use RSD in a distributed environment, a common format for exchanging RSD data structures is needed. The traditional approach would be to use a data stream format. However, this would involve two additional transformation steps whenever RSD data is to be exchanged (internal representation into data stream and back). Since the RSD internal representation has been defined in an object oriented way, this overhead can be avoided, when the complete object is sent across the network.

Today there exists a variety of standards for transmitting objects over the Internet, e.g. *CORBA*, *JavaBeans*, or *Component Object Module COM+*. Since we do not want to commit on either of these, we only define the interfaces of the RSD object class but not its private implementation. This allows others to choose an implementation that fits best to their own data structures. Interoperability between different implementations can be improved by defining translating constructors, i.e. constructors that take an RSD object as an argument and create a copy of it using another internal representation.

6 CCS in Practice

CCS was first used in an industrial setting in the Europort project [8] where large industrial codes were ported to PVM, PARMACS and MPI to run them on massively parallel systems. While we got much positive feedback from the users who praised the stability and versatility of CCS, our resource description language RDL (a predecessor of the RSD described here) was regarded as 'too complex'. Users complained about the tedious task of typing in a long RDL description when they just wanted to run a program on a simple target architecture.

Hence, with the 2^{nd} release of CCS we concealed RDL, giving the user only a simple command line interface. But with the advent of metacomputing, resource description became important again.

Application-Centric Metacomputing. The concept of CCS 4.0 proved useful in two ESPRIT projects [20], both with the goal to provide easy access to industrial applications that are run on Internet or Intranet-connected HPC systems. In both cases, a virtual user access point was implemented in Java that schedules incoming jobs to the temporarily best suited compute server in the Internet. Small and medium enterprises are expected to benefit most by the use of the distributed HPC services for running their most compute-intensive simulation applications – a service they could otherwise not afford due to expensive hardware, maintenance and education cost.

The keyword to these projects is *'application centric metacomputing'*: We do not simply provide raw computing time—as done in several other metacomputing projects—but we rather give access to specific pre-registered applications on a pay-per-use basis. The reasons are twofold:

- First, compute-intensive applications are typically also data-intensive, some of them repetitively running queries against very large databases. Clearly, the databases should be installed prior to access time and updated at night time.

- Second, industrial users are typically not willing to learn about vendor-specific HPC access just to

run their code; they rather prefer to see the machine through their application code's interface.

This scheme was proven in industrial projects running CPU-time intensive CFD simulations on servers in France, Germany and Great Britain. Because CFD simulations produce a large amount of output for visualizing flows and pressures, the server includes a caching facility, allowing the user to specify only that portion of data that is actually needed.

In the second project, we implemented a distributed pharmaceutical application server that allows truely *interactive* design of drug targets. The distributed server contains codes for the prediction of protein functions from sequences, for sensitive sequence searches, for 3D structure generation and for structure comparison. A virtual user access point has been implemented in Java with a job load balancing scheme based on the CIS concept. Security is ensured by data encryption, firewalls and Kerberos authentication. In addition, the server can be installed on in-house LANs for running the most sensitive drug design projects.

7 Related Work

Resource management systems emerged from the need for a better utilization of expensive HPC systems. The *Network Queuing System NQS* [25], developed by NASA Ames for the Cray2 and Cray Y-MP, might be regarded as the ancestor of many modern queuing systems like the Cray *Network Queuing Environment NQE* and the *Portable Batch System PBS.*

Following another path in the line of ancestors, the *IBM Load Leveler* is a direct descendant of *Condor* [27], whereas *Codine* [21] has its roots in *Condor* and *DQS*. They have been developed to support 'high-throughput computing' on UNIX workstation clusters. In contrast to high-performance computing, the goal is here to run a large number of (mostly sequential) batch jobs on workstation clusters without affecting interactive use. The *Load Sharing Facility LSF* [28] is another popular software for utilize LAN-connected workstations for high-throughput computing. For more detailed information on cluster managing software, the reader is referred to [2, 24].

These systems have been extended for supporting the coordinated execution of parallel applications, mostly based on PVM. A multitude of schemes have been devised for high-throughput computing on a somewhat larger scale, including the Iowa State University's *Batrun* [35], the CORBA-based *Piranha* [29], the Dutch *Polder* initiative [11], the *Nimrod* project [1], and the object-oriented *Legion* [22] which proved useful in a nation-wide cluster. While these schemes emphasize mostly on the application support on homogeneous systems, the *AppLeS* project [5] provides application-level scheduling agents on heterogeneous systems, taking into account their actual resource performance.

For the research presented in this paper, the already mentioned *Globus* project [15] is most important. Based on the lessons learned in the I-WAY experiment [14], the National Computational Science Alliance [34] implements a framework of an adaptive wide area metacomputer environment, where Globus, among Condor and *Symbio* (for clustering WindowsNT systems), plays a key role in establishing a national distributed computing infrastructure.

Globus aims at building an adaptive wide area resource environment (AWARE) with a set of tools that enables applications to adapt to heterogeneous and dynamically changing metacomputing environments. Similar to our CIS, a *metacomputing directory service (MDS)* [12] has been proposed to address the need for efficient and scalable access to diverse, dynamic, and distributed information. The API is vendor-independent. MDS is able to handle static and dynamic information. Like our MARS system [19], MDS is intended to manage application specific information that has been found useful in previous program runs (e.g. memory requirements, program structure, communication patterns).

8 Summary

We have presented history, presence and future development of the resource management software CCS. The current release 4.0 has the following features:

- It is modular and autonomous on each layer. New machines, networks, protocols, schedulers, system software, and meta-layers can be added at any point—some of them even without the need to re-boot the system.
- It is reliable. There is no single point of failure. Recovery is done at the machine layer. The center information manager (CIS) is passive and can be restarted or mirrored.
- It is scalable. There exists no central instance. The hierarchical approach allows to connect to other centers' resources. This concept has been found useful in several industrial projects.
- It is extensible. Other resource management systems (e.g. Codine, LSF, Condor) can be linked to CCS without the need to adjust their internal control regime.

From a software engineering view, each module can be implemented in another way, regardless of earlier implementations. From an administrators point of view, the system is easy to administer (by means of RSD and the operator shell), it is reliable, dynamic, and offers customized control on each level.

Compared to the Globus project, there are some similarities, but on a somewhat lower level. With this respect, CCS may be seen as a testbed for gaining valuable experiences with an existing resource management system that provides some important features required for practical metacomputing.

Current Status. Not all of the features have been fully implemented yet. We are currently in the transition phase between the previous RDL language and the here described, more general RSD description tool. Both, the CRM and the CIS have not yet been implemented completely. Furthermore, we plan to change the communication layer from RPCs to MPI-2 or Nexus. These communication layers support the use of multi-threaded daemons, thereby improving the performance of CCS under heavy load.

References

[1] D. Abramson, R. Sosic, J. Giddy, B. Hall. *Nimrod: A Tool for Performing Parameterized Simulations using Distributed Workstations.* 4th IEEE Symp. High Perf. and Distr. Comp. August 1995.

[2] M. Baker, G. Fox, H. Yau. *Cluster Computing Review.* Northeast Parallel Architectures Center, Syracuse University, Nov 1995, New York. http://www.npar.syr.edu/techreports/index.html

[3] B. Bauer, F. Ramme. *A General Purpose Resource Description Language.* Grebe, Baumann (eds), Parallel Datenverarbeitung mit dem Transputer, Springer-Verlag, Berlin, 1991, 68-75.

[4] R. Baraglia, G. Faieta, M. Formica, D. Laforenza. *Experiences with a Wide Area Network Metacomputing Management Tool using IBM SP-2 Parallel Systems.* Concurrency: Practice and Experience, John Wiley & Sons, Ltd., Vol. 8, 1996.

[5] F. Berman, R. Wolski, S. Figueira, J. Schopf, G. Shao. *Application-Level Scheduling on Distributed Heterogeneous Networks.* Supercomputing 96, Nov. 1996.

[6] M. Brune, J. Gehring, A. Reinefeld. *Heterogeneous Message Passing and a Link to Resource Management.* J. Supercomputing, Kluwer Acad. Publ., Vol 11, 355–369 (1997).

[7] M. Brune, J. Gehring, A. Keller, A. Reinefeld. *RSD – Resource and Service Description.* Tech. Rep., Paderborn Center for Parallel Computing, 1998. Also submitted to HPCS'98.

[8] A. Colbrook, M. Lemke, H. Mierendorff, K. Stueben, C.-A. Thole, O. Thomas. *Europort – ESPRIT European Porting Projects.* Int. Conf on High-Perf. Comp. and Netw., Springer LNCS 796 (1994), 46–54.

[9] *Computing Center Software CCS.* Paderborn Center for Parallel Computing. http://www.uni-paderborn.de/pc2/projects/ccs.

[10] T. DeFanti, I. Foster, M. Papka, R. Stevens, T. Kuhfuss. *Overview of the I-WAY: Wide Area Visual Supercomputing.* International Journal of Supercomputer Applications, 10(2):123-130, 1996.

[11] D. Epema, M. Livny, R. van Dantzig, X. Evers, J. Pruyne. *A Worldwide Flock of Condors: Load Sharing among Workstation Clusters.* FGCS, vol. 12, 1996, 53–66.

[12] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, S. Tuecke. *A Directory Service for Configuring High-Performance Distributed Computations.* Proc. 6th IEEE Symp. on High-Performance Distributed Computing 1997

[13] I. Foster, J. Geisler, C. Kesselman, S. Tuecke. *Managing Multiple Communication Methods in High-Performance Networked Computing Systems.* J. Parallel and Distributed Computing, 40:35-48, 1997.

[14] I. Foster, J. Geisler, W. Nickless, W. Smith, S. Tuecke. *Software Infrastructure for the I-WAY High-Performance Distributed Computing Experiment.* Proc. 5th IEEE Symp. on High Performance Distributed Computing, 562-570, 1996.

[15] I. Foster, C. Kesselman. *Globus: A Metacomputing Infrastructure Toolkit.* Journal of Supercomputer Applications.

[16] I. Foster, D. Kohr, R. Krishnaiyer, J. Mogill. *Remote I/O: Fast Access to Distant Storage.* ANL Technical Report.

[17] E. Gabriel, T. Beisel, M. Resch. *PACX (PArallel Computer eXtension), An Installation Guide.* Installation guide for PACX Version 2.0, RUS Tech. Rep., 1997.

[18] J. Gehring, F. Ramme. *Architecture-Independent Request-Scheduling with Tight Waiting-Time Estimations.* IPPS'96 Workshop on Scheduling Strategies for Parallel Processing, 1996, Hawaii, Springer LNCS 1162, 41–54.

[19] J. Gehring, A. Reinefeld. *MARS – A Framework for Minimizing the Job Execution Time in a Metacomputing Environment.* Future Generation Computer Systems, Vol. 12, 87-99 (1996).

[20] J. Gehring, A. Reinefeld, A. Weber, *PHASE and MICA: Application Specific Metacomputing.* Europar'97, Passau, Germany, 1997.

[21] GENIAS Software GmbH. *Codine: Computing in Distributed Networked Environments.* http://www.genias.de/products/codine/.

[22] A. Grimshaw, J.B. Weissman, E.A. West, E.C. Loyot. *Metasystems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems.* J. Par. Distr. Comp. 21 (1994), 257–270.

[23] GRM. *Resource Manager Specification v0.3.* http://www.globus.org/scheduler/grm_spec.html

[24] J.P. Jones, C. Brickell. *Second Evaluation of Job Queueing/Scheduling Software: Phase 1 Report.* Nasa Ames Research Center, NAS Tech. Rep. NAS-97-013, June 1997.

[25] B.A. Kinsbury. *The Network Queuing System.* Cosmic Software, NASA Ames Research Center, 1986.

[26] J. Linn. *Generic Security Service Applications Programming Interface.* Internet RFC 1508, (1993).

[27] M.J. Litzkow, M. Livny. *Condor–A Hunter of Idle Workstations.* Procs. 8th IEEE Int. Conf. Distr. Computing Systems, June 1988, 104–111.

[28] LSF. *Product Overview.* http://www.platform-.com/products/, July 1997.

[29] S. Maffeis. *Piranha: A CORBA Tool for High Availability.* IEEE Computer, April 1997, 59–66.

[30] F. Ramme, T. Römke, K. Kremer. *A Distributed Computing Center Software for the Efficient Use of Parallel Computer Systems.* HPCN Europe, Springer LNCS 797, Vol. II, 129-136 (1994).

[31] F. Ramme. *Transparente und effiziente Nutzung partitionierbarer Parallelrechner.* PhD Dissertation (in German), Paderborn Center for Parallel Computing, 1997.

[32] A. Reinefeld, R. Baraglia, T. Decker, J. Gehring, D. Laforenza, F. Ramme, T. Römke, J. Simon. *The MOL Project: An Open Extensible Metacomputer.* Proceedings HCW'97, Vienna, IEEE Computer Society Press, 17–31.

[33] L. Smarr, C.E. Catlett. *Metacomputing.* Communications of the ACM 35,6(1992), 45–52.

[34] R. Stevens, P. Woodward, T. DeFanti, C. Catlett. *From I-WAY to the National Technology Grid.* Communiations of the ACM 11(1997), 51–60.

[35] F. Tandiary, S.C. Kothari, A. Dixit, E.W. Anderson. *Batrun: Utilizing Idle Workstations for Large-Scale Computing.* IEEE Parallel and Distr. Techn., Summer 1996, 41–48.

Acknowledgments

Thanks to the members of the CCS-team, who have spent a tremendous effort on the development, implementation, and debugging since the project start in 1992: *Bernard Bauer, Matthias Brune, Harald Dunkel, Jörn Gehring, Oliver Geisser, Christian Hellmann, Axel Keller, Achim Koberstein, Rainer Kottenhoff, Karim Kremers, Fru Ndenge, Friedhelm Ramme, Thomas Römke, Helmut Salmen, Dirk Schirmer, Volker Schnecke, Jörg Varnholt, Leonard Voos, Anke Weber.*

Author Biographies

Axel Keller received his diploma in computer science from the University of Paderborn in 1993. As a staff member of the Paderborn Center for Parallel Computing he spent much time in designing and implementing CCS releases 2, 3, and 4.

Alexander Reinefeld received his CS diploma and PhD from the University of Hamburg in 1982 and 1987, respectively. In 1984/85 and 1987/88, he was awarded a DAAD scholarship and a Sir Walton Killam Post Doctoral fellowship for a two years study at the University of Alberta. He worked as a software consultant and as an assistant professor at the University of Hamburg. Since 1992, he manages the Paderborn Center for Parallel Computing.