UNIVERSITY OF CRETE
SCHOOL OF SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

Service Composition and Service−level Agreements in Open Distributed Systems

Manolis Marazakis

Doctoral Dissertation

Heraklion, August 2000

UNIVERSITY OF CRETE
SCHOOL OF SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

Service Composition and Service−level Agreements in Open Distributed Systems

Dissertation submitted by
Manolis Marazakis
in partial fulfillment of the requirements for
the PhD degree in Computer Science

Author:

Manolis Marazakis
Department of COMPUTER SCIENCE

Examination Committee:

Christos Nikolaou, Professor, Dept. of Computer Science, University of Crete, Advisor

Panos Constantopoulos, Professor, Dept. of Computer Science, University of Crete, Reader

Evangelos Markatos, Assistant Professor, Dept. of Computer Science, University of Crete, Reader

Stelios Orphanoudakis, Professor, Dept. of Computer Science, University of Crete, Reader

Dimitris Plexousakis, Assistant Professor, Dept. of Computer Science, University of Crete, Reader

Elias Houstis, Professor, Professor, Purdue University, USA, External Reader

Seif Haridi, Professor, Royal Institute of Technology, Stocholm, Sweden, External Reader

Approved by:

Panos Constantopoulos
Chairman of Graduate Studies

Heraklion, August 2000

Service Composition and Service−level Agreements in Open Distributed Systems

Manolis Marazakis

PhD Thesis

Department of Computer Science
University of Crete

ABSTRACT

The Internet's explosive growth motivates a shift from standalone single−purpose applications to network−centric software systems that consist of independent and widely distributed components. Such systems use open protocols to combine the services of components in the context of workflow processes, with multiple participants utilizing diverse resources to perform tasks with data and temporal dependencies.

Workflow process support is the application domain of workflow management systems, which however, due to their rigid client/server structure and their implicit assumption that they are the sole governing authority in charge of operations, fail to provide the flexibility required in order to support processes involving autonomous participants with a severely limited degree of control over each other's internal state and procedures. In open systems, explicit agreements are required in order to establish communication channels and other necessary arrangements for co−operation. Such agreements should also cover service−level aspects, such as access control policies, exception handling behavior, and performance−related attributes.

This thesis presents the *Aurora* infrastructure, a step towards integrating service−level agreements in a framework that represents resources, tasks and work sessions among autonomous service−provisioning authorities. By shifting the focus from managing component state transitions, as in most previous systems, to managing asynchronous requests for performing services, the *Aurora* infrastructure preserves the autonomy of service providers and facilitates the composition of services without requiring tight coupling of the systems that need to be integrated for composing a service. The *Aurora* infrastructure supports service−level agreements that describe attributes of components related to access restrictions, exception handling capabilities, and expected performance. Service−level agreements are explicitly represented as run−time objects that encapsulate references to components that implement services. Service−level agreement objects mediate all interactions between service providers and their customers, allowing reliable tracking of the interaction state, monitoring of conformance to commitments on service−level attributes, and automated reaction upon detecting deviations from the expected behavior. Support for service−level management in an open distributed run−time environment is the main contribution of this thesis.

The OMG CORBA platform for distributed applications was used as the implementation basis. This platform was extended with a CORBA−based component model, a container execution framework, event notification services, an access control framework, and a logging/monitoring service. A directory service was developed to maintain metadata that describe components made available by independent providers and their service−level attributes. This directory service complements the functionality of the OMG CORBA Interface Repository. A scripting language, HERMES, was developed as an extension of the popular scripting language Tcl. The HERMES interpreter is a tool to assist in component configuration and interconnection, offering primitives for obtaining and managing references to CORBA objects and components, invoking state monitoring and control operations, issuing and tracking asynchronous requests, and managing asycnhronous event notifications. These extensions to the OMG CORBA platform for distributed applications are a requisite step towards an open distributed platform for service composition.

Service−level agreements and the component/container framework are utilized in the development of a work session framework. Work sessions are managed collections of resources made available by autonomous providers, and enforce an access control policy specified by the collection owner. Thus, work sessions provide a resource sharing context that respects the autonomy of providers. Session resources encapsulate references either to components that directly implement services, or to service–level agreement objects that mediate the sequencing of service performance under the terms promised by autonomous service providers. In this framework, work is carried out by issuing asynchronous requests for service performance. Composite tasks that require several resources can be explicitly represented in this framework, allowing for explicit representation of the producer/consumer dependencies between session resources. This work session framework, in conjunction with the component/container framework and the service–level agreement framework, demonstrates a viable alternative to the currently predominant architecture of workflow management systems.

Two case studies demonstrate the functionality offered by the *Aurora* infrastructure. A WWW Link Recommendation service was developed by composition of two popular WWW services, to highlight the requirements of network–centric applications, and pinpoint missing functionality in existing distributed object infrastructures. A case study from the domain of electronic commerce was developed so as to demonstrate the proposed approach in supporting interactions between service providers and customers based on service–level agreements. This case study also provided the basis for a qualitative comparison with the OMG jointFlow framework for workflow based on distributed objects.
Keywords: Service–level Agreements, Open Distributed Systems, Network–centric Applications, Workflow Management.

Supervisor : Christos Nikolaou, Professor
Department of Computer Science
University of Crete

Σύνθεση Υπηρεσιών και Εγγυήσεις Επιπέδου Υπηρεσίας σε Ανοικτά Κατανεμημένα Συστήματα

Μανόλης Μαραζάκης

Διδακτορική Διατριβή

Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

ΠΕΡΙΛΗΨΗ

Η εκρηκτική ανάπτυξη του Διαδικτύου ενθαρρύνει την στροφή από αυτοτελείς εφαρμογές σε συστήματα λογισμικού αποτελούμενα από ανεξάρτητα και ευρέως κατανεμημένα συστατικά τμήματα (components). Τέτοια *δικτυοκεντρικά συστήματα* χρησιμοποιούν ανοικτά πρωτόκολλα για να συνδιάσουν υπηρεσίες που παρέχονται από ξεχωριστά συστατικά τμήματα σε διατάξεις που μπορεί να μην είχαν προβλεφθεί από τους αρχικούς σχεδιαστές τους, στα πλαίσια διεργασιών που εμπλέκουν πολλούς συμμετάσχοντες οι οποίοι χρησιμοποιούν μια ποικιλία πόρων για να διεκπεραιώσουν εργασίες με αλληλεξαρτήσεις σχετιζόμενες με τον καθορισμό της χρονικής ακολουθίας συμβάντων και την ροή δεδομένων.

Το Πρόβλημα της Παροχής Νέων Υπηρεσιών στο Διαδίκτυο

Υπάρχει τεράστια ζήτηση για νέες εφαρμογές, και η ζήτηση αυτή αναμένεται να αυξηθεί καθώς ο αριθμός των ανθρώπων με δυνατότητα πρόσβασης στο Διαδίκτυο συνεχίζει να αυξάνει. Οι περισσότερες από τις υπάρχουσες εφαρμογές στο Διαδίκτυο παρέχουν πρόσβαση σε υπάρχουσες εφαρμογές και πληροφοριακό περιεχόμενο. Ομως, καταγράφεται ένα αυξανόμενοι ενδιαφέρον για εφαρμογές οι οποίες συνδιάζουν περισσότερες από μία υπηρεσίες παροχής και επεξεργασίας πληροφοριών. Τα παρακάτω παραδείγματα είναι αντιπροσωπευτικά αυτής της ολοένα και πιο δημοφιλούς κατηγορίας εφαρμογών:

- Ολοκληρωμένα ταξιδιωτικά πακέτα: Ο στόχος είναι η παροχή πληροφοριών για διαθέσιμες αε– ροπορικές εταιρίες, ξενοδοχεία, γραφεία ενοικιάσεως αυτοκινήτων, και άλλους εξειδικευμένους φορείς παροχής υπηρεσιών. Ενα σημαντικό θέμα είναι ο χειρισμός όλων των απαιτούμενων κρατήσεων μέσω μιας υπηρεσίας επικοινωνίας με τον χρήστη, η καταγραφή της τρέχουσας κατάστασης και η υποστήριξη τροποποιήσεων κατά περίπτωση. Αν και ο κάθε φορέας παρο– χής υπηρεσιών παραμένει αυτόνομος, είναι απαραίτητο να επιτρέπει ελεγχόμενη πρόσβαση σε ορισμένους από τους πόρους που κατέχει και διαχειρίζεται, στα πλαίσια της διεργασίας του συν– διασμού των διαθέσιμων υπηρεσιών. Επιπλέον, για να γίνουν τέτοιες υπηρεσίες ελκυστικές στους πελάτες είναι απαραίτητο να επιτρέπεται υψηλός βαθμός παραμετροποίησης και υποστήριξη αλλαγών της τελευταίας στιγμής, μέσω μιας και μόνο επαφής χρήσης και λαμβάνοντας υπόψιν τους συγκεκριμένους όρους που διέπουν την κάθε επιμέρους κράτηση (μεταξύ άλλων, όροι για τον χειρισμό ακυρώσεων και αλλαγών). Κατά συνέπεια, η ροή της επεξεργασίας των αιτήσεων εξυπηρέτησης καθορίζεται από συμβάντα, αντί να είναι κατά βάση σειριακή.
- Υπηρεσίες παροχής πληροφοριών εξειδικευμένων κατά περιοχή: Ο στόχος είναι να ληφθεί υπόψιν η γεωγραφική θέση του πελάτη για να του προταθούν υπηρεσίες και προϊόντα που μπορεί να χρησιμοποιήσει στη συγκεκριμένη θέση. Για παράδειγμα, μια υπηρεσία τουριστικών πληροφοριών μπορεί να αξιοποιήσει την γνώση της γεωγραφικής θέσης του πελάτη για να προτείνει κοντινά εστιατόρια και πάρκα ψυχαγωγίας. Είναι σημαντικό να σημειωθεί ότι ενώ τα διαθέσιμα προϊόντα κια υπηρεσίες μπορεί να είναι προσπελάσιμα μέσω πολλαπλών καταλόγων από

ανεξάρτητους φορείς, η σύνθετη υπηρεσία οργανώνει και φιλτράρει την διαθέσιμη πληροφορία με βάση κριτήρια σχετιζόμενα με την γεωγραφική θέση του πελάτη, αν και τέτοια λειτουργικότητα μπορεί να μην υποστηρίζεται άμεσα από τους επιμέρους καταλόγους.

- Υπηρεσίες συσχέτισης πληροφοριών: Ο στόχος είναι να συνδιαστούν πληροφορίες από πολλαπλές πηγές ούτως ώστε να δοθεί στους πελάτες μια νέα άποψη για ένα μεγάλο όγκο πληροφοριών. Για παράδειγμα, μια υπηρεσία συσχέτισης πληροφοριών μπορεί να εντοπίζει σελίδες του Παγκοσμίου Ιστού σχετιζόμενες με ένα δοθέν θέμα, να τις ταξινομεί βάσει βαθμολογίας ως προς ορισμένα κριτήρια, και να παράγει προτάσεις προς τους πελάτες που ενδιαφέρονται να συλλέξουν πληροφορίες γύρω από το δοθέν θέμα. Είναι σημαντικό ότι οι πηγές πληροφορίας μπορεί να είναι έξω από την περιοχή ελέγχου του φορέα παροχής υπηρεσιών που είναι υπεύθυνος για την ταξινόμηση. Επιπλέον, οι φορείς παροχής υπηρεσιών που εμπλέκονται στην διεργασία συσχέτισης πληροφοριών μπορεί να μην γνωρίζουν ότι οι πόροι τους χρησιμοποιούνται για την υλοποίηση μιας σύνθετης υπηρεσίας.

- Διαχείριση επιστημονικών πειραμάτων που εμπλέκουν πολλούς ανεξάρτητους οργανισμούς: Ο στόχος είναι να αξιοποιηθούν τα σύνολα δεδομένων κια η υπολογιστική υποδομή πολλαπλών οργανισμών για την διεξαγωγή μελετών που απαιτούν πολύπλοκη επεξεργασία δεδομένων. Η επεξεργασία αυτήυ μπορεί να περιλαμβάνει μετατροπές της μορφής των δεδομένων, και την προώθηση της εξόδου προγραμμάτων αριθμητικής επεξεργασίας και προσομοίωσης σε επόμενους υπολογισμούς σε αλυσίδα. Ενα παράδειγμα είναι η μελέτη του παγκόσμιου κλίματος, που απαιτεί τον συνδιασμό δεδομένων και αριθμητικών μοντέλων από πολλούς ανεξάρτητους αλλά συνεργαζόμενους επιστημονικούς οργανισμούς, σε διάφορες γεωγραφικές θέσεις. Ενα σημαντικό θέμα έιναι ότι οι απαιτούμενοι υπολογισμοί μπορεί να είναι μακράς διάρκειας. Επιπλέον, είναι σημαντικό να είναι εφικτή η παρακολούθηση της προόδου υπολογισμών που βρίσκονται σε εξέλιξη, καθώς και η παρέμβαση με την τροποποίηση παραμέτρων των αριθμητικών μοντέλων. Το σύνθετο τελικό αποτέλεσμα μπορεί να αξίζει πολύ περισσότερο από τα ενδιάμεσα αποτελέσματα των βημάτων που απαιτείται να ολοκληρωθούν. Επίσης, η υποδομή υπολογισμού και επικοινωνιών πρέπει να είναι σε θέση να επιτρέπει πειραματισμό με ενναλακτικές διατάξεις συστημάτων λογισμικού για επεξεργασία δεδομένων και αριθμητική προσομοίωση.

- Επιχειρησιακές διεργασίες που εμπλέκουν πολλούς ανεξάρτητους οργανισμούς: Ο στόχος είναι η ενοποίηση των κεντρικών επιχειρηματικών διεργασιών (business processes) εταιριών που συνεργάζονται στα πλαίσια μιας κοινοπραξίας. Ο συνολικός επιχειρηματικός στόχος είναι να προσφερθούν στους πελάτες υπηρεσίες και προϊόντα που εξαρτώνται από τους συνδιασμένους πόρους και επιχειρηματικές διεργασίες των εταιριών που συμμετέχουν στην κοινοπραξία. Είναι αναγκαίο να υποστηρίζονται και μακρόπνοες συνεργασίες, υπό την μορφή στρατηγικών συμμαχιών, αλλά και βραχυχρόνιες, ευκαιριακές συνεργασίες για την διάρκεια ενός έργου ή ακόμα και μιας δοσοληψίας. Οι επιχειρηματικές διεργασίες που εμπλέκουν πολλούς ανεξάρτητους οργανισμούς είναι εν γένει διαλλεκτικής μορφής, με άλλα λόγια επιτρέπουν ή/και απαιτούν την παρέμβαση των πελατών που ζητούν την εκτέλεση πολλαπλών υπηρεσιών και λαμβάνουν ενδιάμεσα αποτελέσματα τα οποία μπορούν να χρησιμοποιηθούν σε παραπέρα βήματα.

Ενα θεμελιώδες και κρίσιμο χαρακτηριστικό των εφαρμογών που προαναφέρθησαν είναι το γεγονός ότι, αν και απαιτούν συνεργασία για να δοθεί στους πελάτες μια *σύνθετη υπηρεσία*, η *αυτονομία* των φορέων παροχής υπηρεσιών θέτει σοβαρούς περιορισμούς στον βαθμό του ελέγχου που μπορεί να ασκηθεί στις επιμέρους υπηρεσίες. Αυτόνομοι φορείς παροχής υπηρεσιών δεν μπορούν άμεσα να συνδεθούν σε ένα κοινό πλαίσιο, και, κυρίως, δεν μπορεί να υποτεθεί ότι εκθέτουν πληροφορία για την εσωτερική τους κατάσταση με σκοπό να καταστεί εφικτή η ένταξή τους σε ένα ομογενές κατανεμημένο σύστημα. Αντιθέτως, είναι πιο πιθανό να περιμένει κανείς ότι αυτόνομοι φορείς παροχής υπηρεσιών είναι διατεθειμένοι να εκθέσουν μόνο επαφές χρήσης (interfaces) για τους πελάτες τους, που μπορεί να είναι άλλοι φορείς παροχής υπηρεσιών, για να υποστηρίξουν την υποβολή αιτήσεων εξυπηρέτησης και πιθανώς την παρακολούθηση της προόδου τους. Είναι πιο ρεαλιστικό να υποθέσει κανείς μια *ομοσπονδία* από υπηρεσίες που μπορούν να συνδιαστούν κατά περίπτωση, αντί για μια κοινή πλατφόρμα ενοποίησης. Ειδικές επιχειρηματικές συμφωνίες μπορούν να επιτρέψουν σε συνεργαζόμενους φορείς παροχής υπηρεσιών να κανονίσουν ώστε τα εσωτερικά τους συστήματα να συνεργάζονται για τους σκοπούς της παροχής νέων σύνθετων υπηρεσιών. Το μειονέκτημα αυτής της προσέγγισης είναι ότι υπονοεί κατανόηση των όρων συνεργασίας, η οποία όμως δεν είναι πάντοτε σαφής. Επιπλέον, δεν είναι σαφές τι μπορούν οι πελάτες να περιμένουν από μια σύνθετη υπηρεσία, ειδικά στην περίπτωση προβλημάτων επίδοσης και εξαιρέσεων (τόσο σε επίπεδο συστήματος όσο και σε επίπεδο σημασιολογίας). Συγκεκριμένα, για να γίνουν οι νέες υπηρεσίες ελκυστικές για τουσ πελάτες είναι απαραίτητη η δυνατότητα παρακολούθησης του βαθμού συμμόρφωσης στους όρους και τις συνθήκες που διέπουν την ποιότητα των παρεχόμενων υπηρεσιών, ειδικότερα δεσμεύσεις για ιδιότητες του επιπέδου υπηρεσίας όπως περιορισμοί πρόσβασης, χειρισμός εξαιρέσεων, και επίδοση.

Σημασία του Προβλήματος

Καθώς η σύνθεση υπηρεσιών προσδίδει επιπλέον αξία στα υπάρχοντα συστήματα, με το να χρησι– μοποιεί τις δυνατότητές τους σε νέα πλαίσια, η σχεδίαση και ανάπτυξη υποδομών για την σύνθεση υπηρεσιών μπορεί να θεωρηθεί ως ένα αναγκαίο βήμα για την προστασία των τεράστιων επενδύσεων που έχουν πραγματοποιηθεί για την ανάπτυξη των υπαρχόντων συστημάτων λογισμικού. Επιπλέον, η ευρύτατη αποδοχή και ταχύτατη ανάπτυξη του Παγκοσμίου Ιστού ενθαρρύνουν την σύνθεση υπηρεσιών, διότι ο Παγκόσμιος Ιστός γίνεται το καθολικό μέσο πρόσβασης σε υπηρεσίες. Καθώς συνεχώς και περισσότεροι παραδοσιακοί φορείς παροχής υπηρεσιών αναγκάζονται (λόγω ανταγωνιστικών πιέσεων) να αποκτήσουν παρουσία στον Παγκόσμιο Ιστό, αναπτύσσεται μια ισχυρή τάση να χρησιμοποιήσουν τα εσωτερικά τους συστήματα για την εξυπηρέτηση δοσοληψιών με εξωτερικούς φορείς, όπως πελάτες και επιχειρηματικούς συνεργάτες.

Η σύνθεση υπηρεσιών επαυξάνει την αξία των υπαρχόντων υπηρεσιών με το να τις αξιοποιεί σε πλαίσια τα οποία δεν έχουν πλήρως καθοριστεί εκ των προτέρων, ή ακόμα δεν έχουν καν προβλεφθεί. Αυτή είναι η ουσία της δικτυοκεντρικής προσέγγισης στην κατασκευή συστημάτων λογισμικού. Η ιδιότητα αυτή καθιστά την δικτυοκεντρική προσέγγιση ελκυστική για ένα αυξανόμενο αριθμό εφαρμογών από ποικίλα πεδία, όπως το ηλεκτρονικό εμπόριο και η υποστήριξη συνεργασίας στα πλαίσια έργων. Αυτές οι μορφές εφαρμογών συνήθως περιλαμβάνουν διεργασίες ροής εργασίας (*workflow processes*) που εκτείνονται πέρα από τα διοικητικά όρια ενός οργανισμού, και απαιτούν ένα μοντέλο εκτέλεσης που να επιτρέπει αυτονομία των συμμετεχόντων ως προς το πώς αποδέχονται και εξυπηρετούν αιτήσεις για την εκτέλεση εργασιών.

Μειονεκτήματα των Υπαρχουσών Προσεγγίσεων

Η υποστήριξη διεργασιών (*process support*) είναι το πεδίο εφαρμογής των συστημάτων διαχείρισης ροής εργασίας (*workflow management systems*), τα οποία υποστηρίζουν επιχειρηματικές διεργασίες υλοποιόντας τις κατάλληλες ροές δεδομένων και ειδοποιήσεων για συμβάντα μεταξύ των συμμετε– χόντων [GHS95]. Μια επιχειρηματική διεργασία ορίζει βήματα δραστηριότητας για την επίτευξη ενός επιχειρηματικού στόχου και κανόνες για τον καθορισμό της σειράς αυτών των βημάτων. Οι δραστηριότητες (*activities*) μπορεί να περιλαμβάνουν πράξεις πάνω σε δεδομένα που εξάγονται από επιχειρησιακά συστήματα, όπως συστήματα επεξεργασίας δοσοληψιών, βάσεις δεδομένων και συ– στήματα λογιστικής και παρακολούθησης της τρέχουσας κατάστασης, καθώς και αλληλεπιδράσεις με στελέχη που υποστηρίζονται από διάφορα εργαλεία λογισμικού. Για παράδειγμα, η επεξεργασία μιας παραγγελίας στα πλαίσια μιας διεργασίας προμηθειών μπορεί να περιλαμβάνει την επεξεργασία εγγράφων (όπως σημειώματα και φόρμες) από μια αλυσίδα διοικητικού προσωπικού, αλληλεπίδραση με μηχανισμούς διακίνησης μηνυμάτων (όπως ένα σύστημα διαχείρισης ουρών μηνυμάτων) με σκοπό την πρόσβαση και ενημέρωση πληροφοριών τις οποίες διατηρεί και διαχειρίζεται ένα υπάρχον πληροφο– ριακό σύστημα, μια υποδιεργασία υπό τον έλεγχο ενός συστήματος διαχείρισης και προγραμματισμού χρήσης επιχειρησιακών πόρων (*enterprise resource planning system*) για την παρακολούθηση της πα– ραγγελίας, και εξειδικευμένες υποδιεργασίες που διεκπεραιώνονται από εξωτερικούς προμηθευτές και υπεργολάβους (*subcontractors*).

Διακριτικά γνωρίσματα των εφαρμογών διαχείρισης ροής εργασίας είναι [Sch99] η εν γένει μακρά διάρκεια, οι συχνές αλλαγές (λόγω επιχειρηματιών αποφάσεων όπως αναδιοργανώσεις, βελτιστο– ποιήσεις διεργασιών, και ανάθεση εργασιών σε εξωτερικούς συνεργάτες), η εξάρτηση σε υπάρχοντες επιχειρησιακούς πόρους (που περιλαμβάνουν και ανθρώπους και συστήματα λογισμικού), οι ισχυρές απαιτήσεις για παρακολούθηση της εκτέλεσης βάσει του μοντέλου της διεργασίας, κατανομή πόρων καια εργασίας που μπορεί να διαπερνά τα διοικητικά όρια ενός οργανισμού, και η αλληλεπίδραση με πληθώρα χρηστών που κάνει την ανάθεση πόρων σε εργασίες κρίσιμο πρόβλημα. Εν γένει, τα συστήματα διαχείρισης ροής εργασίας παρέχουν κάποιο μηχανισμό μοντελοποίσης και παράστα–σης διεργασιών, που έχει συνήθως την μορφή ενός γραφήματος δραστηριοτήτων που αναπαριστά τις εξαρτήσεις μεταξύ δραστηριοτήτων, και ένα περιβάλλον εκτέλεσης μέσα στο οποίο οι διεργασίες ενεργοποιούνται και διεκπεραιώνονται με το να ενεργοποιούνται δραστηριότητες, να παρέχονται σε αυτές τα απαιτούμενα δεδομένα εισόδου, και τα παραγόμενα δεδομένα εξόδου να προωθούνται προς τις δραστηριότητες που έπονται αυτών στο γράφημα εξαρτήσεων δραστηριοτήτων.

Μια θεμελιώδης, και συχνά όχι σαφώς καταγεγραμμένη, υπόθεση στην αρχιτεκτονική τύπου πελάτη/εξυπηρέτη (*client/server*) που χαρακτηρίζει την τρέχουσα γενία συστημάτων διαχείρισης ροής εργασίας, όπως ορίζεται από τις προδιαγραφές που δημοσιεύει ο οργανισμός *Workflow Management Coalition* (WfMC) [Hol95], είναι ότι η μηχανή εκτέλεσης της ροής εργασίας (*workflow execution engine*) είναι ο κύριος και κατεξοχήν εξουσιοδοτημένος συντονιστής της εκτέλεσης διεργασιών. Οι διεργασίες θεωρείται ότι είτε περικλείονται εξ'ολοκλήρου στην σφαίρα ελέγχου της μηχανής εκτέλεσης της ροής εργασίας, είτε εκτελούνται υπό τον έλεγχο μηχανών εκτέλεσης της ροής εργασίας που συνεργάζονται μέσω κατάλληλης επαφής χρήσης που υποστηρίζει διαλειτουργικότητα (*interoperability*). Συνεπώς, είναι απαραίτητο να παρασταθεί λεπτομερώς το οργανωτικό μοντέλο ενός οργανισμού κάνοντας χρήση του μηχανισμού μοντελοποίσης και παράστασης διεργασιών που υποστηρίζεται από το σύστημα διαχείρισης ροής εργασίας, και όλες οι εφαρμογές που μπορεί να χρησιμοποιηθούν σε μια ροή εργασίας πρέπει να τροποποιηθούν ώστε να επιτρέπουν ενεργοποίηση και έλεγχο από την μηχανή εκτέλεσης της ροής εργασίας. Συνεπώς, αλλαγές και εν γένει εξέλιξη των διεργασιών συνεπάγονται σημαντικό κόστος ανάπτυξης, εδικά εάν οι διεργασίες απαιτούν την ολοκλήρωση πόρων από εξωτερικούς οργανισμούς. Με το να στηρίζονται σε ένα "μονολιθικό" εξυπηρέτη που είναι υπεύθυνος για τον συντονισμό της ροής εργασίας αλλά και για την εκτέλεση των δραστηριοτήτων, τα υπάρχοντα συστήματα διαχείρισης ροής εργασίας (όπως περιγράφονται από τις προδιαγραφές της WfMC) πάσχουν από σοβαρούς περιορισμούς στην ευελιξία (*flexibility*) και την δυνατότητα κλιμάκωσης (*scalability*). Η αναφορά [PPC97a] εκθέτει τις συνέπειες της άκαμπτης δομής των υπαρχόντων συστημάτων διαχείρισης ροής εργασίας. Ενα σημαντικό πρόβλημα είναι ότι δεν είναι δυνατό να μοιράζονται οι λίστες εργασιών (*work lists*) μεταξύ ετερογενών μηχανών εκτέλεσης της ροής εργασίας, διότι οι λίστες εργασιών δεν είναι προσπελάσιμες έξω από την μηχανή εκτέλεσης της ροής εργασίας. Συνεπώς, για να μπορεί κάποιος να συμμετέχει σε διεργασίες που εμπλέκουν διαφορετικούς εξυπηρέτες, πρέπει ο κάθε εξυπηρέτης να διατηρεί ξεχωριστή λίστα εργασιών, και, επιπλέον, πρέπει οι εφαρμογές–πελάτες να διατηρούν ταυτόχρονα περισσότερες από μια συνδέσεις με εξυπηρέτες. Ενα ακόμα σοβαρό πρόβλημα είναι ότι οι προδιαγραφές της WfMC ορίζουν τρεις διαφορορετικές επαφές χρήσης για την ανάθεση εργασίας σε ανθρώπους, εφαρμογές, και διεργασίες που εκετελούνται σε άλλους εξυπηρέτες, αντίστοιχα. Αυτή η έλλειψη διαφάνειας στο πώς τελικά υλοποιούνται οι δραστηριότητες στα πλαίσια μιας διεργασίας δυσχεραίνει την επανανάθεση εργασίας (*delegation*), καθώς η εφαρμογή ροής εργασίας επιβάλλει ουσιαστικά συγκεριμένο τρόπο για την υλοποίηση της κάθε δραστηριότητας.

Αυτά τα μειονεκτήματα επιτείνονται σε ανοικτά περιβάλλοντα συστημάτων όπως το Διαδίκτυο, όπου διασυνδεόμενα και αλληλεξαρτώμενα συστατικά αναμένεται να είναι σε θέση να χειριστούν αλληλε–πιδράσεις που δεν ακολουθούν προδιαγεγραμμένους περιορισμούς χρονοπρογραμματισμού (*scheduling constraints*). Οι απαιτήσεις σεβασμού της αυτονομίας των διαφόρων φορέων συνεπάγονται ότι δεν μπορεί κανείς να θεωρήσει δεδομένες βολικές υποθέσεις και ρυθμίσεις για τα κανάλια επικοινωνίας, την εξω–τερικά προσπελάσιμη λειτουργικότητα, τις πολιτικές ελέγχου πρόσβασης, την αναμενόμενη επίδοση, και την συμπεριφορά κατά τον χειρισμό εξαιρέσεων. Αντίθετα, τέτοιες υποθέσεις και ρυθμίσεις πρέπει να τεκμηριωθούν σαφώς, ανά περίπτωση και ξεχωριστά για κάθε διαθέσιμο πόρο. Με δεδομένους τους σοβαρούς περιορισμούς στην πρόσβαση σε εσωτερική πληροφορία κατάστασης και μηχανισμούς ελέγχου των πόρων που διαχειρίζονται ανεξάρτητοι φορείς παροχής υπηρεσιών, οι εφαρμογές ροής εργασίας πρέπει να γίνουν αρκετά ευέλικτες ώστε να μπορούν να χειριστούν συμφωνίες επιπέδου υπηρεσίας (*service–level agreements*), οι οποίες ορίζουν τους αμοιβαία αποδεκτούς όρους και συνθήκες που διέπουν την αλληλεπίδραση ανάμεσα σε ένα πελάτη και μια υπηρεσία, μέσα σε σαφώς ορισμένα όρια ελέγχου που σέβονται την εξουσία και αυτονομία των ανεξάρτητων φορέων που συμμετέχουν σε διεργασίες.

Η εισαγωγή συμφωνιών επιπέδου υπηρεσίας επιβάλλει την αναθεώρηση της συνολικής οργάνωσης και του μοντέλου εκτέλεσης των εφαρμογών ροής εργασίας, λόγω της αυτονομίας των φορέων παροχής υπηρεσιών κια της απαίτησης για σαφή περιγραφή, παρακολούθηση, και έλεγχο/εξασφάλιση ιδιοτήτων επιπέδου υπηρεσίας. Οι υποδομές κατανεμημένων εφαρμογών οφείλουν να εστιάσουν περισσότερο στην διαχείριση αιτήσεων εξυπηρέτησης και την αξιόπιστη καταγραφή της προόδου τους, αντί για την καταγραφή των μεταβολών κατάστασης δραστηριοτήτων. Η εξέλιξη αυτή κρίνεται αναγκαία λόγω του διαλλεκτικού χαρακτήρα των δραστηριοτήτων που εμπλέκουν αυτόνομους φορείς παροχής υπηρεσιών. Οι δραστηριότητες διαλλεκτικού χαρακτήρα δεν υποστηρίζονται επαρκώς από τα υπάρχοντα μοντέλα διεργασιών που υποθέτουν ότι οι δραστηριότητες ενεργοποιούνται *άπαξ*, ξεκινούν τον κύκλο εκτέλεσής τους, και δεν παράγουν αποτελέσματα παρά μόνο όταν αυτός τερματιστεί είτε επιτυχώς είτε ανώμαλα λόγω εξαιρέσεων ή βλαβών. Οι υπάρχουσες προσεγγίσεις [GHS95, Hol95] δεν υποστηρίζουν άμεσα την ακύρωση μιας αίτησης εξυπηρέτησης που βρίσκεται σε εξέλιξη ή την επανόρθωση/αποκατάσταση (*compensation*) των συνεπειών της. Μια και οι υπάρχουσες προσεγγίσεις δεν παρσιτάνουν άμεσα τις αιτήσεις εξυπηρέτησης και τις συμφωνίες επιπέδου υπηρεσίας ως αντικείμενα πρώτης τάξεως, το πρόβλημα της υποστήριξης σύνθετων διαλλεκτικών πρωτοκόλλων ανάμεσα σε πελάτες και φορείς παροχής υπηρεσιών είναι έξω από τις προδιαγραφές των υπαρχόντων προσεγγίσεων. Επιπλέον, οι υπάρχουσες προσεγγίσεις δεν υποστηρίζουν επαρκώς την παρακολούθηση και τον έλεγχο σε πραγματικό χρόνο των αιτήσεων εξυπηρέτησης. Οι ελλείψεις αυτές επιβάλλουν σημαντικές επεκτάσεις στις υπάρχουσες πλατφόρμες κατανεμημένων εφαρμογών.

Τεχνικές Προκλήσεις

Η σύνθεση υπηρεσιών από γεωγραφικώς κατανεμημένους ανεξάρτητους φορείς αποτελεί ειδική περίπτωση του προβλήματος της διαλειτουργικότητας σε καθολική κλίμακα (*global–scale interoperability*), και θέτει δύσκολες τεχνικές προκλήσεις:

- Αυτονομία: Το Διαδίκτυο είναι ένα ανοικτό, κατανεμημένο σε ευρεία κλίμακα σύστημα που δεν ανήκει και δεν ελέγχεται από μια και μόνο αρχή. Οι ανεξάρτητοι φορείς παροχής υπηρεσιών δεν εκθέτουν λεπτομέρειες των εσωτερικών τους διεργασιών, καθώς η γνώση αυτή θα μπορούσε να αποκαλύψει τα μοναδικά τους πλεονεκτήματα και μειονεκτήματα σε ανταγωνιστές. Είναι διατεθειμένοι να εκθέσουν μόνο αφαιρετικές περιγραφές των υπηρεσιών που παρέχουν, ενδεχομένως επαυξημένες με πληροφορίες κια μηχανισμούς ελέγχου ιδιοτήτων του επιπέδου υπηρεσίας. Η αυτονομία συνεπάγεται επίσης την ετερογένεια, με άλλα λόγια την ύπαρξη διαφορετικών εννοιολογικών μοντέλων, μοντέλων διεργασιών και εκτέλεσης για τους ανεξάρτητους φορείς παροχής υπηρεσιών. Για τον σεβασμό της αυτονομίας των φορέων παροχής υπηρεσιών, είναι απαραίτητο η υποδομή υποστήριξης εφαρμογών να εξασφαλίζει ότι οι συμμετέχοντες σε διεργασίες δεν χρειάζεται να έχουν άμεση γνώση των μοντέλων διεργασιών και θεμάτων υλοποίησης για άλλους συμμετέχοντες. Θα πρέπει να στηρίζονται μόνο σε αφαιρετικές περιγραφές των διαθέσιμων υπηρεσιών, όπως οι επαφές χρήσης των υπηρεσιών, και σε μια γενικής εφαρμογής υποδομή για ελεγχόμενη αλληλεπίδραση μεταξύ πελατών και φορέων παροχής υπηρεσιών.

- Κατανομή ευρείας κλίμακας: Η ευρείας κλίμακας κατανομή των πόρων και των συστατικών τμημάτων λογισμικού που υλοποιούν τις υπηρεσίες εισάγει μεταβλητές (ενδεχομένως μεγάλες) καθυστερήσεις, αυξημένη πιθανότητα για μερικές βλάβες και εξαιρέσεις, και την ανάγκη υποστήριξης ενός μεγάλου αριθμού υπολογιστικών κόμβων με διαφορετικές δυνατότητες και επιδόσεις, διασυνδεόμενων μέσω συνδέσμων επικοινωνίας με ποικίλες χωρητικότητες. Αυτά τα χαρακτηριστικά των ανοικτών κατανεμημένων συστημάτων τονίζουν την σημασία των απαιτήσεων για αξιόπιστη παρακολούθηση της προόδου των αιτήσεων εξυπηρέτησης, και επιβάλλουν ένα μοντέλο εκτέλεσης στηριζόμενου στην διαχείριση ασύγχρονων αιτήσεων εξυπηρέτησης.

- Αυξανόμενες αλληλεξαρτήσεις μεταξύ συστημάτων: Καθώς η επεξεργασία αιτήσεων εξυπηρέτησης μπορεί να εμπλέκει πόρους υπό των έλεγχο περισσοτέρων της μιας αρχών, οι σύνθετες υπηρεσίες εισάγουν εξαρτήσεις με πολύπλοκες δομές μεταξύ των φορέων παροχής υπηρεσιών. Κάθε φορέας παροχής υπηρεσιών ελέγχει, και συνεπώς μπορεί να θεωρηθεί άμεσα υπεύθυνος για, τα τμήματα εκείνα στην επεξεργασία μιας αίτησης εξυπηρέτησης που πραγματοποιούνται μέσα στα όρια της περιοχής ελέγχου του. Με το να στηρίζεται, συχνά έμμεσα, και σε άλλους φορείς παροχής υπηρεσιών, το θέμα της αξιόπιστης παρακολούθησης της προόδου των αιτήσεων εξυπηρέτησης και των αμοιβαίων δεσμεύσεων μεταξύ συνεργαζόμενων φορέων αποκτά ιδιαίτερη βαρύτητα. Η επανανάθεση εργασίας σε ειδικευμένους φορείς παροχής υπηρεσιών γίνεται η συνήθης πρακτική, εισάγοντας επιπλέον πολυπλοκότητα στην παρακολούθηση των εξαρτήσεων. Δεν είναι σπάνιο διαφορετικοί συμμετέχοντες σε μια διεργασία παροχής σύνθετης υπηρεσίας να έχουν διαφορετική αντίληψη των ορίων αλληλεπίδρασης, καθώς οι φορείς που παρέχουν τις επιμέρους υπηρεσίες δεν είναι υποχρεωμένοι να αποκαλύπτουν τις εξαρτήσεις τους ή τις σχέσεις επανανάθεσης εργασίας που έχουν αναπτύξει με άλλους φορείς.

Διατύπωση της Ερευνητικής Θέσης της Διατριβής

Για να διευκολυνθεί η αναχρησιμοποίηση και ο συνδιασμός υπηρεσίων από ανεξάρτητες αρχές, είναι ουσιώδες να υποστηριχθεί ο αυτοματισμός της εφαρμογής συμφωνιών επιπέδου υπηρεσίας, και η αξιόπιστη καταγραφή της κατάστασης αλληλεπίδρασης. Η υποστήριξη συμφωνιών επιπέδου υπηρεσίας απαιτεί κατάλληλη πλατφόρμα εφαρμογών.

Η παρούσα διατριβή παρουσιάζει την υποδομή *Aurora*, μια κατανεμημένη πλατφόρμα υποστήριξης εφαρμογών που συνδιάζουν συστατικά τμήματα λογισμικού που διατίθενται από αυτόνομους φορείς παροχής υπηρεσιών. Η πλατφόρμα αυτή σχεδιάστηκε και αναπτύχθηκε ως επέκταση της πλατφόρμας κατανεμημένων αντικειμένων OMG CORBA [COR94]. Ενα χαρακτηριστικό γνώρισμα αυτής της πλατ– φόρμας, που την διαφοροποιεί από προηγούμενες εργασίες, είναι η σαφής καταγραφή των δεσμεύσεων από ανεξάρτητους φορείς παροχής υπηρεσιών για ιδιότητες του επιπέδου υπηρεσίας. Η καταγραφή επιτυγχάνεται με την εφαρμογή ενός πλαισίου για συμφωνίες επιπέδου υπηρεσίας, οι οποίες τεκμηριώ– νουν την αναμενόμενη συμπεριφορά των υπηρεσιών, όσον αφορά τους περιορισμούς πρόσβασης, τον χειρισμό εξαιρέσεων, την αυτόματη αντίδραση σε συμβάντα όπως αποκλίσεις από τους συμφωνηθέντες όρους, και την αναμενόμενη επίδοση.

Το Σχήμα 0.1 δείχνει τον ρόλο της υποδομής *Aurora* ως κοινής βάσης για ποικίλες εφαρμογές με κατανομή ευρείας κλίμακας. Το Σχήμα 0.2 δείχνει την θέση της υποδομής *Aurora* στην στοίβα

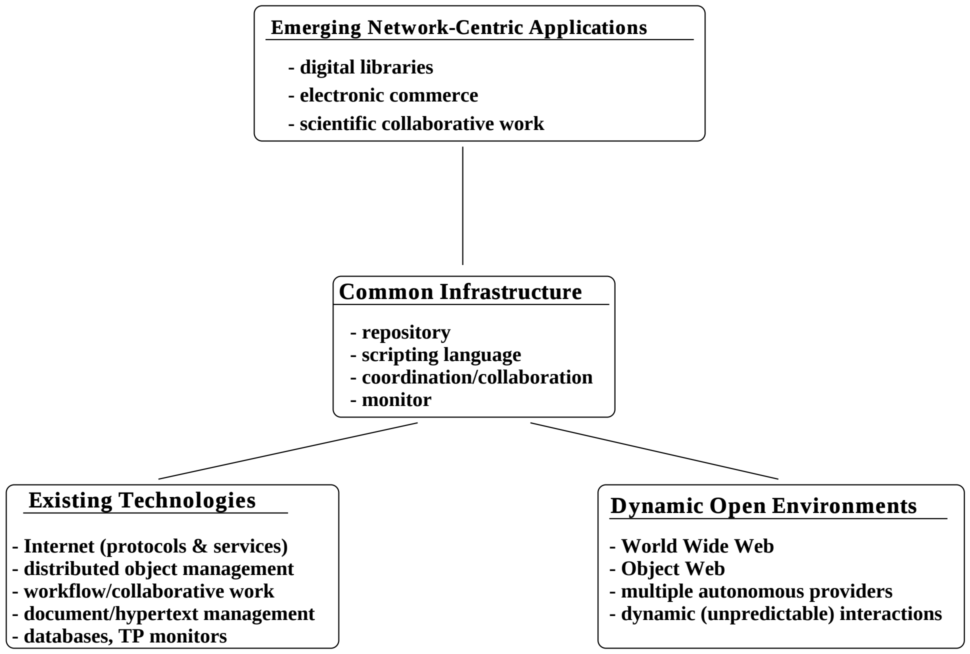

Emerging Network-Centric Applications
- digital libraries
- electronic commerce
- scientific collaborative work
Common Infrastructure
- repository
- scripting language
- coordination/collaboration
- monitor
Existing Technologies
- Internet (protocols & services)
- distributed object management
- workflow/collaborative work
- document/hypertext management
- databases, TP monitors
Dynamic Open Environments
- World Wide Web
- Object Web
- multiple autonomous providers
- dynamic (unpredictable) interactions

Figure 0.1: Προς μια Κοινή Υποδομή για Δικτυοκεντρικές Εφαρμογές.

πρωτοκόλλων. Πρόκειται για τεχνολογία τύπου *middleware* [Ber96], βασισμένη στην γλώσσα προ– γραμματισμού Java [GJS96] και την πλατφόρμα κατανεμημένων αντικειμένων OMG CORBA [COR94]. Προσφέρει αντικειμενοστραφή πλαίσια και υπηρεσίες υποστήριξης για να βοηθήσει στην ανάπτυξη δικτυοκεντρικών εφαρμογών, υποστηρίζοντας την διαχείριση της ροής αιτήσεων εξυπηρέτησης, δε–

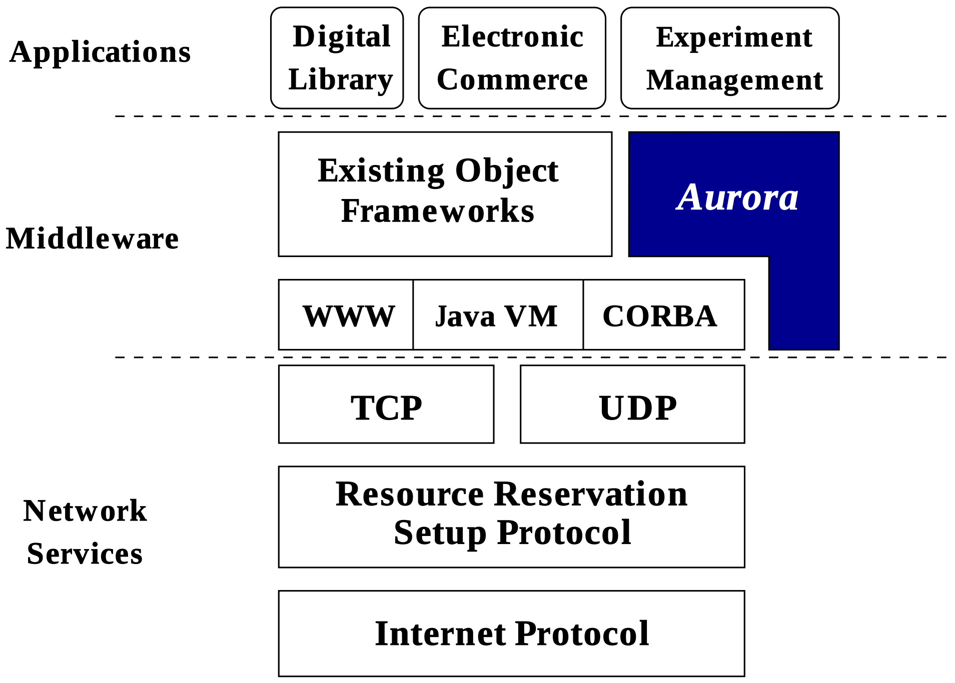


Figure 0.2: Η Θέση της Υποδομής *Aurora* στην Στοίβα Πρωτοκόλλων.

δομένων, κια ειδοποιήσεων για συμβάντα ανάμεσα σε κατανεμημένα συστατικά τμήματα λογισμικού που διατίθενται από ανεξάρτητους φορείς παροχής υπηρεσιών. Οπως περιγράφεται λεπτομερώς στην διατριβή, η υποδομή *Aurora* σέβεται την αυτονομία των φορέων παροχής υπηρεσιών, και επιπλέον υποστηρίζει την σαφή παράσταση συμφωνιών επιπέδου υπηρεσίας, και την εξέταση και τον έλεγχό τους σε πραγματικό χρόνο.

Σύνοψη Ερευνητικών Συνεισφορών

Ακολουθώντας την αναφορά [MMWFF92], το πλαίσιο διαχείρισης συμφωνιών επιπέδου υπηρεσίας που παρουσιάζεται στην διατριβή αυτή αποδίδει μεγαλύτερη βαρύτητα στην αντιμετώπιση της ροής εργασίας ως διαδικασίας για την ικανοποίηση στόχων και δεσμεύσεων (*satisfaction−centered perspective*), αντί ως διαδικασία για την παραγωγή προϊόντων πληροφορίας (*production−centrered perspective*). Υπό το πρίσμα αυτό, οι δεσμεύσεις, οι συνθήκες ικανοποίησης και η έγκαιρη περάτωση είναι οι βασικές κατευθυντήριες γραμμές. Μια επιπλέον διαφοροποίηση από την σύνηθη αντίληψη για την ροή εργασίας που εστιάζει στα δεδομένα και τον χειρισμό τους (data−centric perspective) [BDS+93] είναι ότι η παρούσα διατριβή εστιάζει στην διαχείριση ασύγχρονων αιτήσεων εξυπηρέτησης. Η διατριβή περιγράφει ένα κατανεμημένο περιβάλλον εκτέλεσης (βασισμένο στην πλατφόρμα OMG CORBA) για συστατικά τμήματα λογισμικού που αναπαριστούν διεκπεραιωτές εργασιών, οι οποίοι παρέχουν υπηρεσίες σε πελάτες. Αυτό το περιβάλλον εκτέλεσης συνδιάζει συστατικά τμήματα λογισμικού, που υλοποιούνται συναρμολογώντας κατανεμημένα αντικείμενα, με μια σειρά από υπηρεσίες υποστήριξης, ειδικά με ένα περιβάλλον εκτέλεσης που λειτουργεί ως κέλυφος για συστατικά τμήματα λογισμικού και χειρίζεται τις χαμηλού επιπέδου λεπτομέρειες της υποβολής και παρακολούθησης της προόδου αιτήσεων εξυπηρέτησης. Αυτό το περιβάλλον−κέλυφος (container run−time environment) παρέχει την βάση για ένα πλαίσιο που αναπαριστά πόρους και εργασίες, επιτρέποντας τον συνδιασμό τους σε αυθαίρετες διαμορφώσεις με την χρήση μηχανισμών σύνδεσης συστατικών τμημάτων λογισμικού. Αυτό το πλαίσιο για κατανεμημένες συνεδρίες εργασίας (distributed work sessions) υλοποιείται πάνω από την βασισμένη στην πλατφόρμα OMG CORBA υποδομή για συστατικά τμήματα λογισμικού που αντιστοιχούν σε υπηρεσίες που ανήκουν και λειτουργούν υπό την εποπτεία ανεξάρτητων φορέων παροχής υπηρεσιών.

Οι συνεισφορές της ερευνητικής προσπάθειας που παρουσιάζεται στην παρούσα διατριβή είναι οι εξής:

1. Ενα πλαίσιο για συμφωνίες επιπέδου υπηρεσίας σε ανοικτά κατανεμημένα συστήματα: Οι συμ–φωνίες επιπέδου υπηρεσίας αναπαριστώνται άμεσα στο περιβάλλον εκτέλεσης, μεσολαβούν στις αλληλεπιδράσεις μεταξύ φορέα παροχής υπηρεσίων και πελάτη, παρακολουθούν την συμμόρ–φωση ή την απόκλιση από δεσμεύσεις για ιδιότητες επιπέδου υπηρεσίας, καταγράφουν με τρόπο αξιόπιστο τις ενέργειες του πελάτη και του φορέα παροχής υπηρεσίων, και αυτομάτως ενερ–γοποιούν διαδικασίες για την επανόρθωση αποκλίσεων από την αναμενόμενη συμπεριφορά της υπηρεσίας.

2. Μια υποδομή (βασισμένη στην πλατφόρμα OMG CORBA) για συστατικά τμήματα λογισμικού που υλοποιούν υπηρεσίες, οι οποίες ανήκουν εξ'ολοκλήρου και ελέγχονται αποκλειστικά από τους αντίστοιχους φορείς παροσχής υπηρεσιών: Η υποδομή αυτή περιλαμβάνει μια υπηρεσία κατα–λόγου για περιγραφικές πληροφορίες (μετα–δεδομένα) για τις λειτουργικές επαφές χρήσης και τις συμφωνίες επιπέδου υπηρεσίας που υποστηρίζουν διάφορα συστατικά τμήματα λογισμικού, ένα περιβάλλον–κέλυφος για συστατικά τμήματα λογισμικού και ασύγχρονες αιτήσεις εξυπη–ρέτησης, υπηρεσίες ειδοποίησης για συμβάντα, υπηρεσίες καταγραφής και παρακολούθησης συμβάντων, ένα πλαίσιο ελέγχου πρόσβασης, και ένα διερμηνευτή μιας γλώσσας για την διευ–κόλυνση της διαμόρφωσης συστατικών τμημάτων λογισμικού και της διαχείρισης ασύγχρονων αιτήσεων εξυπηρέτησης.

3. Ενα πλαίσιο για κατανεμημένες συνεδρίες εργασίας με υποστήριξη για ελεγχόμενη κοινή πρόσβαση σε πόρους: Οι κατανεμημένες συνεδρίες εργασίας εισάγονται ως ένα πλαίσιο για ελεγχόμενη πρόσβαση σε κοινόχρηστους πόρους, υποστηρίζοντας την επιβολή περιορισμών προσπέλασης και χρήσης από τους ανεξάρτητους φορείς παροχής υπηρεσιών. Οι μετέχοντες σε συνεδρίες εργασίας συντονίζουν την ροή εργασιών υποβάλλοντας αιτήσεις εξυπηρέτησης, για την επεξεργασία των οποίων απαιτείται η χρήση κοινόχρηστων πόρων. Σύνθετες εργασίες αναπαριστώνται ως πόροι με εξαρτήσεις τύπου παραγωγού/καταναλωτή από άλλους πόρους.

4. Δύο συστήματα επίδειξης:

   - Μια προσπελάσιμη μέσω του Παγκοσμίου Ιστού υπηρεσία που προτείνει συνδέσμους για δοθείσα θεματική κατηγορία, συνδιάζοντας τις πολύ δημοφιλείς υπηρεσίες Yahoo και Altavista
   - Ενα περιβάλλον ηλεκτρονικού εμπορίου που υποστηρίζει συμφωνίες επιπέδου υπηρεσίας κατά την επεξεργασία παραγγελιών από πελάτες

Acknowledgements

The research presented in this dissertation are the results of almost years of full−time effort. During this period, I had the benefit of working and interacting with several people that helped me get through the ordeal of graduate school. To these people I will always be indepted.

First of all, I would like to thank my advisor, Professor Christos Nikolaou for finding the time and energy to supervise my research efforts, both at the MSc and the PhD level, and for laying out for me the standards and guidelines for serious research in the area of distributed software systems. For this guidance, and his tolerance to my occasional lapses, I will always be grateful.

The Institute of Computer Science (ICS/FORTH) supported my research, by providing access to equipment, by providing financial support, and by subsidizing the cost of my participation in international conferences and workshops. The system administrators at ICS/FORTH, Panos Sikas, Panos Psikos, and Vaggelis Karagiannis, helped me get through several “idiosynchrancies” of the computing equipment. Likewise, the system administrators at the Department of Computer Science, University of Crete, Mrs Maria Mamalaki and Mrs Cristina Vaglini, managed to make ends meet in several crisis situations during the crucial last six months of the implementation effort described in this dissertation. The Department’s Graduate Programme secretary, Mrs Rena Kalaitzaki, helped in making necessary arrangements in time.

During my studies at the Department of Computer Science, University of Crete, I had the priviledge to interact with several members of the faculty, both as student and as teaching assistant. Among others, I had the priviledge to attend classes taught by, and work as teaching assistant for, Dr Christos Nikolaou, Dr Giorgos Tziritas, and Dr Evangelos Markatos. They all set high standards for academic performance and integrity, which I can only wish to be able to emulate.

The members of the examining committee for this thesis provided valuable feedback about the initial (inadequate) draft of the text. In particular, the comments by Dr Panos Constantopoulos and Dr Stelios Orfanoudakis led to a major revision of this text. Dr Dimitris Plexousakis provided detailed technical comments, as well as insightful questions about theoretical aspects of the *Aurora* infrastructure presented in this dissertation. The external readers, Dr Seif Haridi and Dr Elias Houstis also provided valuable comments and suggestions for improvements in the presentation of the research described in this dissertation. Any remaining omissions or errors (hopefully minor !) are entirely my responsibility.

I am particularly grateful to Dr Evangelos Markatos for his invaluable technical comments and his crucial assistance in preparing for the public defense of this thesis. I consider myself to be extremely lucky for having the chance to discuss matters related to distributed systems technologies, during an extended period of time, with such a dedicated researcher and teacher as Dr Markatos.

I would also like to express my gratitude to all former and current members of the Parallel and Distributed Systems (PLEIADES) [1] division at ICS/FORTH. I am particularly indepted to Mrs Penelope Constanta for her words of encouragement, and her invaluable assistance in numerous administrative, financial, as well as technical matters. I would also like to thank Dr Petros Kavassalis for allowing me to participate in the preparation of a major European project.

Special thanks are due to the members of `The Cage`, an informal group of people that at some time or other worked in, or regularly visited, room `G171` in the PLEIADES laboratory. To these people I wish good luck in their pursuits, and I hope that eventually we will be able to get together again to talk about things from our past in `The Cage`.

The ideas about open distributed systems expressed in this dissertation matured through several heated discussions with Dimitris Papadakis and Giorgos Georgiannakis, people with unusually good listening skills as well as in−depth knowledge of systems, literature, history, and culture. I wish for them every happiness

[1] `http://www.ics.forth.gr/pleiades`

and success in their current endeavours. The same wish goes for Stavros Papadakis, a fellow student of exceptional technical skill and moral quality. Dimitris, Giorgos, and Stavros proved to be reliable friends in both good times and hard times, and I only hope that we remain friends for life.

The completion of the work described in this dissertation would not have been possible without the unfailing support of my parents, Lambros and Calliope, and my fiance, Romina.

Contents

Dissertation Abstract i

Περίληψη Διδακτορικής Διατριβής iii

Acknowledgements xi

Contents xiv

List of Tables xv

List of Figures xvii

1 Introduction 1
- 1.1 The Problem of Providing New Services Over the Internet . . . 1
- 1.2 Problem Significance . . . 3
- 1.3 Shortcomings of Existing Approaches . . . 3
- 1.4 Technical Challenges . . . 5
- 1.5 Thesis Statement . . . 6
- 1.6 Summary of Contributions . . . 7
- 1.7 Organization of Dissertation . . . 8

2 Motivation and Outline of Approach 11
- 2.1 Adding Value to Existing Services by Composition . . . 11
- 2.2 Importance of Having a Common Request Protocol . . . 12
- 2.3 Implications of Autonomy on Perceived Service Levels . . . 13
- 2.4 Effects of Large−Scale Distribution . . . 13
- 2.5 Service Level Management through Explicit Agreements . . . 13
- 2.6 Infrastructure . . . 14
- 2.7 Composition via Scripting . . . 15
- 2.8 A Note on Transactional Properties . . . 16

3 Related Work 19
- 3.1 Interoperability . . . 19
- 3.2 Distributed Workflow Enactment . . . 21
- 3.3 Service Contracts . . . 24
- 3.4 Service Level Management . . . 25

4 Component/Container Framework 29
- 4.1 Component Model . . . 29
  - 4.1.1 Basic Concepts of the Component Model . . . 30
  - 4.1.2 Management of Exported Interface References . . . 30
  - 4.1.3 Management of Required Interface References . . . 30
  - 4.1.4 Management of Event Notifications . . . 31
  - 4.1.5 Component Metadata . . . 31
  - 4.1.6 Component Installation and Support Services . . . 31

4.1.7 The ComponentProxy Interface 32
4.1.8 Support for On−line Monitoring and Control of Components 33
4.2 Management of Asynchronous Requests 33
4.2.1 Asynchronous Requests in CORBA 34
4.2.2 Primitives for Asynchronous Request Management 35
4.2.3 Callbacks for Notifications about Asynchronous Requests 37
4.3 Access Control Framework 38
4.4 Support for Event−Driven Execution 39
4.5 The HERMES Scripting Language 41
4.6 A Note on Performance Issues 43

5 Service level Agreements **47**
5.1 Definition of Relationship Boundaries 47
5.2 Logging and Monitor Infrastructure 49
5.3 Integration with the Component/Container Framework 53

6 Work Session Framework **55**
6.1 Resources, Tasks and Work Sessions 56
6.2 Differences from Other Approaches 57

7 Case Study **65**
7.1 Description of Components 65
7.2 Order Processing Workflow 66
7.3 Integration using the *Aurora* Framework 67
7.4 Integration using the OMG jointFlow Framework 68
7.5 A Demonstration of Service−level Agreements 68

8 Conclusions and Perspective **73**
8.1 Summary of Results 73
8.2 Directions for Further Research 74

References **76**

Appendix A: IDL Specification for the Aurora Component Framework **83**

Appendix B: IDL Specification for the Aurora Container Run−time Environment **89**

Appendix C: IDL Specification for the Aurora Service−Level Agreement Framework **96**

Appendix D: IDL Specification for the Aurora Work Session Framework **100**

Appendix E: The OMG jointFlow Specification **105**

Appendix F: IDL Specification for Electronic Commerce Case Study **108**

List of Tables

4.1 Management Interfaces Exported by Aurora Components. 29
4.2 Methods of the `ComponentControl` interface. 33
4.3 Asynchronous request management primitives. 36
4.4 Asynchronous request management callbacks. 38
4.5 HERMES/Tcl Commands for ECA rules and their Parameters. 41

List of Figures

0.1 Προς μια Κοινή Υποδομή για Δικτυοκεντρικές Εφαρμογές. viii
0.2 Η Θέση της Υποδομής *Aurora* στην Στοίβα Πρωτοκόλλων. ix

1.1 Towards a Common Infrastructure for Network−Centric Applications. 6
1.2 The *Aurora* Infrastructure in the Protocol Stack. 7

2.1 Composition of two popular WWW services. 12
2.2 Overview of *Aurora* Architecture. 14

4.1 Structure of a container hosting ComponentProxy objects. 34
4.2 States in the processing of a work request. 34
4.3 Interaction with a Component Hosted within a Container. 44
4.4 Access Control in the Context of a Work Session. 44
4.5 Example of composite event notifications by combining the `SubjectGroupFSM`, `TIME scheduler`, `SubjectGroupFilter`, and `EventLoop` services. 45

5.1 Elements of the service−level agreement interface. 48
5.2 Structure of the *Aurora* Monitor. 54
5.3 Service−level agreements as entry points to services. 54

6.1 A Satisfaction−Oriented View of Workflow. 61
6.2 Instantiation and Control of Process Instances in the *Aurora* run−time environment. 61
6.3 Elements of the *Aurora* session framework. 62
6.4 Example of a work session involving autonomous participants. 63
6.5 The Workflow Management Coalition Reference Model. 63

7.1 Order life−cycle in the e−commerce case study 66
7.2 Service−level agreement for shipping an order: Normal case. 69
7.3 Service−level agreement for shipping an order: Exception handling. 71

Some inspiring words . . .

In pioneer days, they used oxen for heavy pulling, and when one ox couldn't budge a log they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers. —— Grace Murray Hopper (1906 – 1994).

What is it indeed that gives us the feeling of elegance in a solution, in a demonstration? It is the harmony of the diverse parts, their symmetry, their happy balance; in a word it is all that introduces order, all that gives unity, that permits us to see clearly and to comprehend at once both the ensemble and the details. —— Henry Poincare (1854 – 1912).

Prediction is very difficult, especially about the future. —— Niels Bohr (1885 – 1962).

To Lambros, Calliope, and Romina.

Chapter 1

Introduction

The Internet's explosive growth motivates a shift from stand−alone applications towards software systems consisting of independent and widely distributed computational components. Such *network−centric systems* use open protocols to combine the services offered by individual components in configurations that may not have been foreseen by their developers, in the context of *processes* involving multiple participants that utilize a variety of resources to perform tasks with interdependencies related to temporal sequencing and data flow.

1.1 The Problem of Providing New Services Over the Internet

There is a tremendous demand for new applications, and this demand is expected to increase as the number of people with Internet access continues to grow. Most current Internet applications provide access to existing back−end applications and information content. However, there is a growing interest in applications that combine several information processing services. The following examples are representative of this increasingly popular class of applications:

- Integrated travel packages: The aim is to provide information about available airlines, hotels, car rental agencies, and other specialized service providers. An important issue is to handle all the required reservations through a single front−end service, and provide tracking information and support for on−demand customization. Although each service provider remains autonomous, it is required to allow controlled access to certain of their resources, in the context of the process that combines the available services. Moreover, for making such services attractive to customers it is essential to allow them a high degree of customization and support for last−minute changes, through a single interface and taking into account the specific terms for each reservation (including the terms for handling cancellations and changes). Thus, the flow of request processing is event−driven, rather than sequential.
- Location−specific information services: The aim is to take the customer's location into account in recommending services and products that the customer can use at the specified location. For example, a tourist information service can use location information to recommend nearby restaurants and amusement parks. It is important to note that while available products and services may be accessible through multiple catalogues from independent providers, the composite service organizes and filters

the available information using location–related criteria, although such functionality may not be directly supported by the individual catalogues.

- Information correlation services: The aim is to combine information from multiple sources so as to provide customers with new insight regarding a large volume of information. For example, an information correlation service could locate WWW pages related to a specified topic, rank them using some criteria, and generate recommendations to customers who are interested in obtaining information on the specified topic. It is important that the information sources may be beyond the scope of control of the service provider that performs the ranking. Moreover, the service providers involved in the correlation process need not be aware that their resources are being used for realizing a composite service. An example of an information correlation service is presented in detail in Section 2.

- Multi–organization experiment management: The aim is to utilize the data sets and computational infrastructure of multiple organizations in performing complex processing of data sets. Such processing may involve performing data format translations, and feeding the output of simulation programs to other computations in pipeline. An example is a global climate study, that combines data and numerical models from multiple organizations, at various geographic locations. An important issue is that computations may be of long duration. Furthermore, it is essential to be able to monitor the progress of ongoing computations, as well as to be able to manipulate model parameters. The composite result is more valuable than the intermediary results of the steps involved. The underlying infrastructure should facilitate experimentation with alternative configurations of data processing and numerical simulation software systems.

- Multi–organization business processes: The aim is to integrate the core business processes of enterprises that co–operate in the context of a coalition. The overall business objective is to offer to customers services and products that rely on the combined resources and business processes of multiple enterprises participating in the coalition. It is necessary to support both long–term collaborations, taking the form of strategic alliances, and short–term, opportunistic collaboration for the duration of a single project or transaction. Multi–organization business processes are usually *conversational*, in other words allow or require interaction where customers issue multiple service invocations and receive intermediate results that may be used in further interactions. The execution time–line of a composite service includes one or more service requests, as well as feedback and control operations during the ongoing service performance and the delivery of service results.

A crucial characteristic of such applications is that, while co–operation is required for providing customers with a *composite service*, the *autonomy* of service providers severely restricts the degree of control that can be exercised on individual services. Autonomous service providers cannot be readily coupled in a common framework, and, more importantly, cannot be assumed to expose internal state information for allowing tight coupling in a coherent distributed system. Rather, it is more likely to expect that autonomous service providers will be willing to expose only interfaces for customers, or other service providers, to submit requests and possibly to track their progress. Instead of a single integration platform, it is more realistic to assume a *federation* of services that can be combined on a case–by–case basis. Although special business deals could allow service providers to arrange for their internal systems to co–operate for the purposes of providing new composite services, such an approach has the disadvantage of relying on an *implicit* understanding on the terms of co–operation. Moreover, it is not clear what customers can expect from a composite service, especially in the event of performance problems and exceptions (both system–level and semantic). In particular, for new services to become attractive for customers it is essential to be able to monitor conformance to terms and conditions for the qualities of services, in particular commitments about *service–level* attributes such as access restrictions, exception handling, and performance.

1.2 Problem Significance

Since service composition adds value to existing systems, by using their capabilities in new contexts, the design and development of platforms for service composition can be seen as a requisite step for protecting the tremendous investments that have been made in developing existing single–purpose software systems. Moreover, the wide acceptance and rapid growth of the World Wide Web motivate service composition, since the Web becomes the pervasive service access medium. As more and more traditional service providers are forced (due to competition pressures) to establish a presence on the Web, there is a strong potential for their internal systems to be used in serving transactions from external entities, such as customers and business partners.

Composition of services adds value to existing services by utilizing them in contexts that have not been fully prescribed or even anticipated. This is the essence of *network–centric* approach to constructing software systems, which makes it attractive for a growing number of applications from diverse domains, such as electronic commerce and support for collaborative work. This type of applications typically involves workflow processes that span administrative boundaries, and necessitate an execution model that allows for the autonomy of participants in how they accept and perform requests for work.

1.3 Shortcomings of Existing Approaches

Process support is the application domain of *workflow management systems* that support *business processes* by realizing appropriate flows of data and event notifications among participants [GHS95]. A business process defines activity steps for achieving a business goal and rules for determining the sequence of these steps. Activities may involve manipulations of operational data extracted from line–of–business systems, such as transaction processing systems, databases and accounting/tracking systems, as well as interactions with human workers that are assisted by various tools. For example, order processing in a procurement process may involve processing of documents (such as memos and forms) by a chain of administration personnel, interaction with message brokering facilities such as a queuing system in order to access and update information managed by a legacy information system, a sub–process managed by an enterprise resource planning system for tracking the order, and specialized sub–processes executed by external suppliers and subcontractors.

Distinctive characteristics of workflow applications include [Sch99] long duration, frequent changes (due to business–level decisions such as reorganizations, process optimization and outsourcing), reliance on existing business assets (both human workers and software systems), strong requirements for monitoring execution in terms of the underlying process model, distribution that may span administrative domains, and interaction with several end–users that makes assignment of work to resources a crucial issue. Typically, workflow management systems provide some type of process modeling mechanism, most commonly a form of activity network that represents the dependencies between activities using an activity graph model, coupled with an execution environment within which processes can be enacted by initiating activities, passing them their required input data, and forwarding their output data to their successors in the activity graph.

A fundamental, and often implicit, assumption underlying the client/server architecture of current–generation workflow management systems, as outlined in the specifications published by the Workflow Management Coalition (WfMC) [Hol95], is that the workflow execution engine is the principal and authoritative co–ordinator of processes, which are either completely contained within its sphere of control or execute under the control of another workflow engine that supports interoperation by implementing an appropriate interoperability interface. Therefore, an enterprise's organizational model needs to be represented in detail using the system's modeling functionality and all applications involved in the workflow need to be adapted to allow invocation and control by the workflow engine. Changes and evolution of processes can therefore entail major development efforts, especially if they involve the integration of resources from external organizations. By relying on a monolithic server that is responsible for both workflow coordination and activity execution, current workflow management systems as defined by the WfMC impose severe limitations on flexibility and scalability. [PPC97a] outlines the consequences of the current systems' rigid structure. A major problem is that work lists cannot be shared by heterogeneous workflow engines, since they are not externally accessible. Thus, in order for a participant to take part in multiple workflows on different workflow servers, a separate work list needs to be maintained at each server and client applications need to maintain multiple dedicated connections. Moreover, three different interfaces are used for assigning work to human participants, invoked applications, and sub–workflows on other servers. This lack of transparency in how the activities are implemented makes delegation of work difficult, as the workflow application mandates the actual activity implementation.

Such shortcomings are exacerbated in open systems such as the Internet where interconnected and interdependent components are expected to be able to handle interactions that do not adhere to predefined scheduling constraints. Autonomy considerations imply that assumptions and convenient arrangements about communication channels, exported functionality, access policies, performance, and exception handling behavior cannot be taken for granted, but rather have to explicitly established, in a case–by–case fashion for each resource of interest. With severely limited access to the internal state and operational procedures of independent service providers, workflow applications need to become flexible enough so as to accommodate *service–level agreements* that define the mutually accepted terms and conditions for interaction, within well–defined boundaries of control that respect the authority of independent participants.

The introduction of service–level agreements necessitates rethinking the overall organization and execution model of workflow applications, due to the autonomy of service providers and the requirement for explicit specification, monitoring, and enforcement of service–level qualities. The distributed application infrastructure needs to focus more on the management of requests and reliable tracking of their progress, rather than tracking the state transitions of activities. This is necessary in order to capture the *conversational* nature of activities that involve autonomous service providers.

Conversational activities cannot be adequately supported by existing process models which assume that activities are invoked once, begin execution, and produce no results until they are completed successfully or stopped due to exceptions or failures. In particular, existing approaches [GHS95, Hol95] do not directly support the cancellation of a service request in progress, or compensation for its effects. Since current approaches typically do not explicitly represent requests and service−level agreements as first−class objects, the problem of supporting a complex conversational protocol between a customer and a service provider is beyond the scope of existing approaches. What is more, existing approaches do not adequately support on−line monitoring and control of service requests. Such shortcomings necessitate major extensions to existing application platforms.

1.4 Technical Challenges

The composition of services from geographically distributed independent providers, a special case of the problem of global−scale interoperability, poses hard technical challenges:

- Autonomy: The Internet is an open, large−scale distributed system that is not owned and controlled by any single authority. Independent service providers do not expose details of their internal processes, since such knowledge could reveal their unique strengths and weaknesses to competitors. They are only willing to expose functional abstractions of their available services, possibly augmented with information and controls on service−level qualities. Autonomy also implies *heterogeneity*, in other words different conceptual, process, and/or execution models for the independent service providers. In order to respect the autonomy of service providers, the application infrastructure should ensure that process participants require no direct knowledge of the process models and implementations used by other participants. Instead, they can rely only on functional abstractions such as published interfaces for available services, and generic infrastructure support for controlled interaction between customers and providers of services.
- Large−scale distribution: The large−scale distribution of the resources and software components that implement services introduces variable (possibly long) delays, an increased likelihood of partial failures, and the need to accommodate a large number of hosts with varying capabilities, over communication links with various capacities. These characteristics of open distributed system environments stress the requirements for reliable tracking of requests, and necessitate an execution model that facilitates *asynchronous request management*.
- Increasing interdependencies among systems: Since the processing of service requests may involve resources under the control of multiple authorities, composite services introduce complex dependency structures among service providers. Each service provider can control, and be held directly responsible for, segments of service processing that take place within its domain of control. By relying, often indirectly, on other service providers, it becomes essential to be able to perform reliable tracking of the progress of requests, and the mutual commitments between collaborating service providers. Delegation of work to specialized providers is becoming standard practice, further complicating dependency tracking. It is often the case that different participants in the process of providing a composite service have differing views of the

boundaries of interaction, since providers do not necessarily reveal their dependencies or delegation relationships with other providers.

1.5 Thesis Statement

In order to facilitate reuse and combination of services from independent authorities, it is essential to support automatic sequencing of service−level agreements, and reliable tracking of interaction state. An appropriate platform needs to be in place to enable service−level agreements.

This dissertation describes the *Aurora* infrastructure, a distributed component platform for applications that combine components made available by autonomous service providers. This platform has been designed and implemented as an extension of the OMG CORBA distributed objects platform [COR94]. A distinguishing aspect of this infrastructure is the *explicit* representation of the commitments made by independent service providers. This is achieved by a framework for *service−level agreements* that document the expected behavior of services, in terms of access restrictions, exception handling, automated reaction to events such as deviations from agreed−upon terms, and performance.

Figure 1.1 illustrates the role of the *Aurora* infrastructure as a common infrastructure for diverse large−scale distributed applications. Figure 1.2 places the *Aurora* infrastructure in

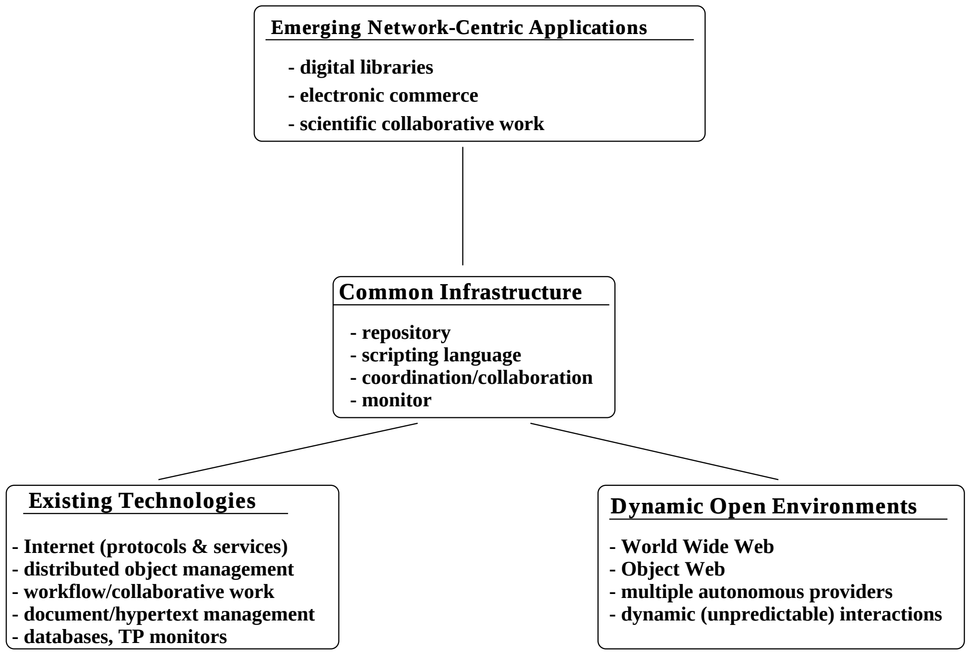

Emerging Network-Centric Applications
- digital libraries
- electronic commerce
- scientific collaborative work
Common Infrastructure
- repository
- scripting language
- coordination/collaboration
- monitor
Existing Technologies
- Internet (protocols & services)
- distributed object management
- workflow/collaborative work
- document/hypertext management
- databases, TP monitors
Dynamic Open Environments
- World Wide Web
- Object Web
- multiple autonomous providers
- dynamic (unpredictable) interactions

Figure 1.1: Towards a Common Infrastructure for Network−Centric Applications.

the protocol stack. It is a *middleware technology* [Ber96], based on the Java programming language [GJS96] and the OMG CORBA platform for distributed objects [COR94], that offers object frameworks and support services for assisting developers in the construction of network−centric applications by managing the flow of requests, data, and event

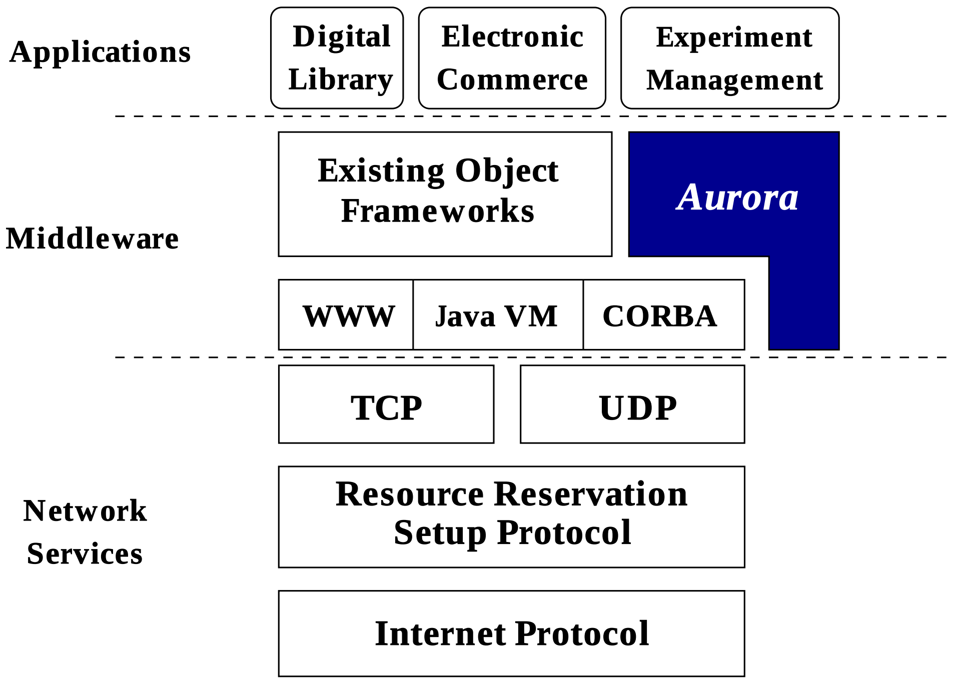


Figure 1.2: The *Aurora* Infrastructure in the Protocol Stack.

notifications among distributed software components from independent service providers. As described in detail in later chapters, the *Aurora* infrastructure respects the autonomy of service providers, but still enables the explicit representation, and on−line inspection and control, of service−level agreements.

1.6 Summary of Contributions

Following [MMWFF92], the service−level management framework presented in this dissertation shifts the emphasis from a *production−centered* view of workflow towards a *satisfaction−centered* perspective, where commitments, conditions of satisfaction and timely completion are the guiding concerns. As a further departure from a data−centric perspective on workflow [BDS$^+$93], we focus on the management of asynchronous work requests, and describe a CORBA−based distributed run−time environment for hosting components that represent *performers* of work providing services to *customers*. This run−time environment combines components, implemented as assemblies of distributed objects, with a range of support services and in particular a *container* execution environment that handles the low−level details of issuing and monitoring the progress of requests. This provides the basis for a framework that represents resources and tasks, allowing for their combination in arbitrary configurations using component linking facilities provided by the underlying component model. This framework for *distributed work sessions* is built on top of a CORBA−based platform for components that implement services, owned and managed exclusively by their respective service providers.

The contributions of the research presented in this dissertation are as follows:

1. A framework for service−level agreements in open distributed systems: Service−

level agreements are represented as first–class objects in the run–time environment, to mediate the interaction between a service provider and a customer, monitor conformance to commitments on service–level attributes, reliably log the actions of customer and service provider, and automatically invoke compensating actions in the event of deviations from the expected behavior.

2. A CORBA–based platform for components that implement services owned and managed exclusively by their respective service providers: This platform includes a directory service that maintains metadata describing the functional interfaces as well as the service–level agreements supported by software components, a container run–time environment for hosting components and supporting asynchronous requests, event notification services, logging/monitoring services, an access control framework, and a scripting language interpreter for facilitating component configuration and asynchronous request management.

3. A work session framework for controlled resource sharing: Work sessions are introduced as a resource sharing context that enables the enforcement of access restrictions in using resources from independent providers. Session participants sequence work assignments by issuing requests to be served by using resources. Composite tasks are represented as resources with producer/consumer dependencies with other resources.

4. Two demonstrator systems:

    - A WWW link recommendation service that combines services from the Yahoo directory and the Altavista search engine
    - An electronic commerce environment that supports service–level agreements in the processing of orders placed by customers.

1.7 Organization of Dissertation

In this dissertation we present a framework and infrastructure services for managing service–level agreements for workflow in open distributed systems. Chapter 2 presents a WWW link recommendation service that was developed as a demonstrator to highlight important aspects of workflow management in an open distributed environment with autonomous service providers. This demonstrator system serves to motivate the development of the *Aurora* infrastructure presented in later chapters of this dissertation. Chapter 3 presents a review of related work, in order to highlight the distinguishing aspects of the work presented in this dissertation. The service–level management framework relies on a container/component framework, presented in Chapter 4, that was developed as an extension of the OMG CORBA platform for distributed applications [COR94], along with a number of services added to this platform in order to support access control restrictions and event–driven interactions between autonomous partners. These extensions and additional services serve to provide the implementation basis for *work sessions*, which represent a workspace for controlled access to shared resources by autonomous participants in dynamic workflows. Chapter 5 presents the service–level management framework. Chapter 6 presents the work session framework, which allows service providers to *publish* their resources and specify appropriate access restrictions for customers that submit *work requests* and initiate tasks that may involve interdependent steps. This framework supports the notion of a *shared workspace* that allows controlled sharing of resources from autonomous providers, offering a workflow paradigm that is appropriate for open distributed environments.

A case study in Chapter 7 serves to demonstrate the proposed approach in supporting interactions between service providers and customers based on service–level agreements. This case study also provided the basis for a qualitative comparison with the OMG jointFlow framework [jFl98, Sch99], an adaptation of the WfMC run–time reference model to a business objects execution environment. Chapter 8 concludes this dissertation with a brief summary of the research results presented in this dissertation, and suggestions for further research. Appendices A–F present the interface specifications for the component/container framework, the service–level agreement framework, and the work session framework that comprise the *Aurora* infrastructure. These specifications, expressed using the OMG IDL language, supplement the functional descriptions of the *Aurora* infrastructure presented in Chapters 4–6.

Chapter 2

Motivation and Outline of Approach

As an example of the effects of autonomy on composition, consider the following scenario, inspired from [Ude]. The Yahoo directory [1] provides links to WWW sites organized in a hierarchy of categories [2]. The AltaVista search engine [3] can provide the number of references to a specified WWW link as recorded in its database of links (through the `link:` directive). For a collection of WWW links on similar topics, such as those listed under the same Yahoo category, the number of references can be interpreted as a measure of the relative popularity of the corresponding WWW sites, assuming that WWW sites with a higher number of references are recognized as more authoritative references for material that falls under the specified category.

Based on this intuition, we have implemented a simple workflow (see Figure 2.1) where the two services are combined to provide a ranking of WWW sites in a given category based on their respective number of references, after discarding links whose reference count is below a given threshold. Thus a simple−minded but useful recommendation service is built, by presenting the user with a *top−N* list of WWW links according to the ranking, and optionally fetching the HTML pages referenced by these links, to generate a briefing page by simply concatenating the contents of the original pages. This simple example illustrates several important aspects of workflow in an open environment, and in particular the impact of autonomy in the realization of a workflow.

2.1 Adding Value to Existing Services by Composition

First of all, the example illustrates that composition of components in a distributed system adds value to existing services by utilizing them in contexts that have not been fully prescribed or even anticipated. This is the essense of the *network−centric* approach to constructing software systems, which makes it attractive for a growing number of applications from diverse domains, such as electronic commerce and support for collaborative work. This type of applications typically involves workflows that span administrative boundaries, and necessitate an execution model that allows for the autonomy of participants in how they accept and perform requests for work. In this simple example, the workflow involves two participants, other than the end−user, which are fully autonomous service providers that independently maintain their services and make them available for use under their own terms, through interfaces of their own choosing.

[1] `http://dir.yahoo.com`

[2] An example of a Yahoo category is $/Science/Astronomy/$, which contained 3248 links, on August 16, 1999

[3] `http://www.altavista.com`

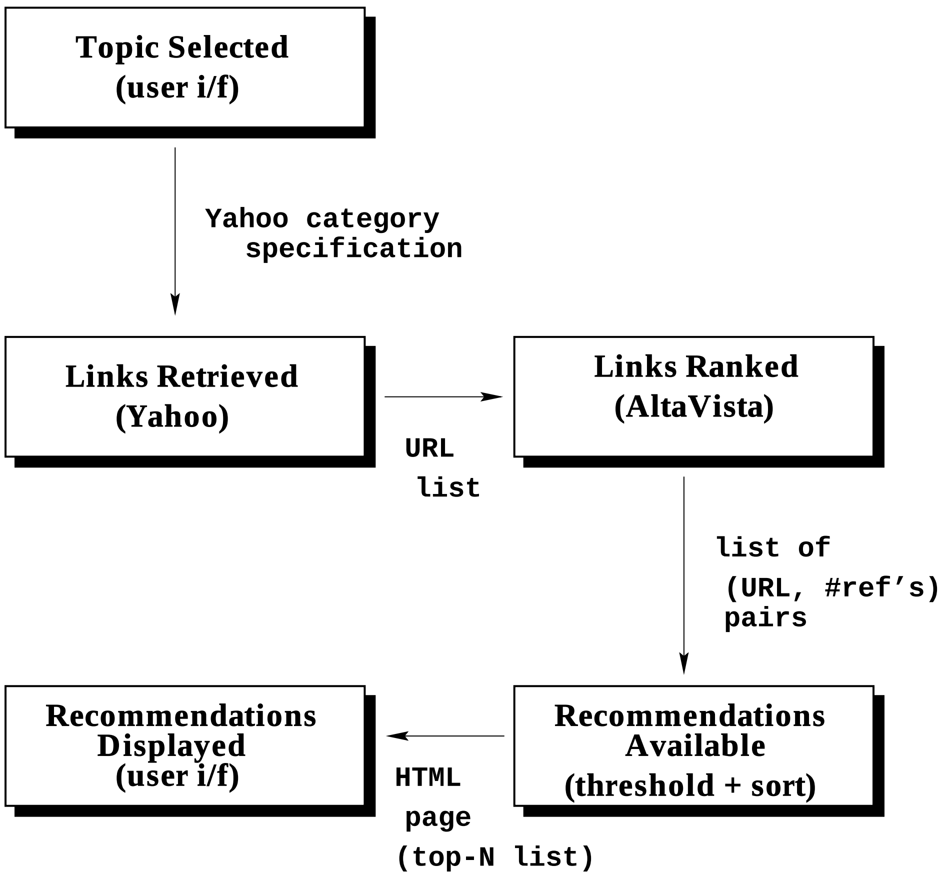

Topic Selected (user i/f)
Yahoo category specification
Links Retrieved (Yahoo)
URL list
Links Ranked (AltaVista)
list of (URL, #ref's) pairs
Recommendations Available (threshold + sort)
HTML page (top-N list)
Recommendations Displayed (user i/f)

Figure 2.1: Composition of two popular WWW services.

2.2 Importance of Having a Common Request Protocol

In this example the request protocol for both service providers is HTTP, allowing the end−user to combine their services despite the fact that they do not expose their internal state and procedures. Complications would arise if their request protocols were different. In such a case "proxy" objects could insulate the end−user from the differences in the protocols, and expose a uniform request protocol, independent of the underlying services. A common request protocol to be implemented by all process participants allows a clean separation between workflow coordination actions and actual work performance. Such an explicit distinction has important consequences for workflow management in an open environment. Different participants can share heterogenous and autonomous service providers while imposing minimal requirements on the providers, namely that they publish a description of their services, and support the common request protocol either natively or though proxy objects. Moreover, delegation of tasks is enabled without requiring the client to become aware of this arrangement between service providers. This can be achieved by allowing a service provider to forward requests, possibly after decomposition and some pre−processing, to other co−operating providers. Optionally, the provider can inform the client that a request has been forwarded through a callback invocation mechanism. Using callbacks, a provider can explicitly accept or deny to process a request under the terms specified by the client, or initiate a negotiation process over the terms specified by the client.

2.3 Implications of Autonomy on Perceived Service Levels

In this example access is unrestricted due to the nature of the services involved, and absolutely no guarantees are promised by the providers regarding performance, reliability, and other service−level qualities of their services (such as accuracy of the results produced). In business process settings it is expected that the terms and conditions of access to services provided by partners will be subject to explicit agreements regarding access restrictions as well as service−level attributes. Workflows act as service requesters and participants (humans, apps, organizations) act as service providers to workflows. A workflow has a−priori no authority over service providers invoked during the course of its execution. Service−level agreements define a service provider's degree of exposure of internal state and procedures and the specific control and monitoring actions offered to service users. This view necessitates making service−level agreements first−class objects in the system.

2.4 Effects of Large−Scale Distribution

Another aspect of composing the services of wide−area distributed components is that the workflow is subject to variable (possibly long) delays and higher chances of partial failures introduced by large−scale distribution. This aspect of open environments makes the commonly used blocking requests unattractive. In this example, extracting the reference counts from AltaVista for all the links in a Yahoo category may take several minutes, during which the end−user has no control over the progress of the ongoing (aggregate) request. Such long−duration interaction necessitate asynchronous request management, where, instead of issuing a single blocking request and waiting for results or an failure/exception signal, the client carries out a sequence of steps to initiate a request (such as binding to the service provider, setting parameters, and indicating the method to be executed) and arranges with the service provider to receive results and status information in an asynchronous manner, whenever such information becomes available. This is a shift from a state−based view of the workflow towards a transaction−oriented view, as requests are treated as first−class objects rather than being simple messages exchanged between workflow participants. A consequence of long process duration is that requests need to have persistent unique identifiers whose lifetime exceeds that of the application programs that generated them, so as to minimize the coupling between a *work performer* and its *customers*.

2.5 Service Level Management through Explicit Agreements

It is important to note that current transaction processing techniques [GR93] cannot be directly applied, since there may be multiple related interactions between any two partners over a time interval of unpredictable duration, introducing a much greater variability in response time than commonly assumed in transaction processing applications, thereby making locking protocols for data consistency unattractive. Furthermore, the autonomy of participants and the heterogenity of their internal processes complicate consistency management. In the context of a process, all messages (requests, responses, asynchronous event notifications) are semantically meaningful and therefore cannot be simply discarded or rolled−back in the event of deviation from the desired course. An alternative to classic transaction processing is needed to provide guarantees of some form, allow monitoring of conformance to agreed terms and conditions, and handle deviations in a meaningful manner, without imposing restrictions on the autonomy of participants.

Such requirements fall within the domain of contract law, which governs common business practices such as cancellations, modifications, and compensation for requests that have been issued. In our work we attempt to reify a form of service−level contracts in a framework supporting configurations of resources, process participants, and tasks in flexible combinations.

2.6 Infrastructure

This dissertation presents a distributed software component infrastructure thta serves as the basis for implementing work sessions combining services made available from autonomous service providers. This infrastructure, *Aurora*, provides a CORBA−based component model, a set of component management services, and support for autonomous services to export explicit contracts on the terms and conditions for usage by clients. In particular, the component framework supports dynamic management of dependencies among components, on−line monitoring and control of ensembles of interconnected components, run−time inspection and manipulation of configuration properties and component interconnections, and enforcement of access restrictions and constraints on acceptable performance and exception handling. The work session and service level agreement frameworks that we propose are built on top of this component model to provide a controlled run−time environment for workflow management applications that can cope with the large−scale distribution and autonomy of service providers. Figure 2.2 illustrates the basic components of the *Aurora* infrastructure. The *Aurora* infrastructure was first presented in [NMP+97, MPN97].

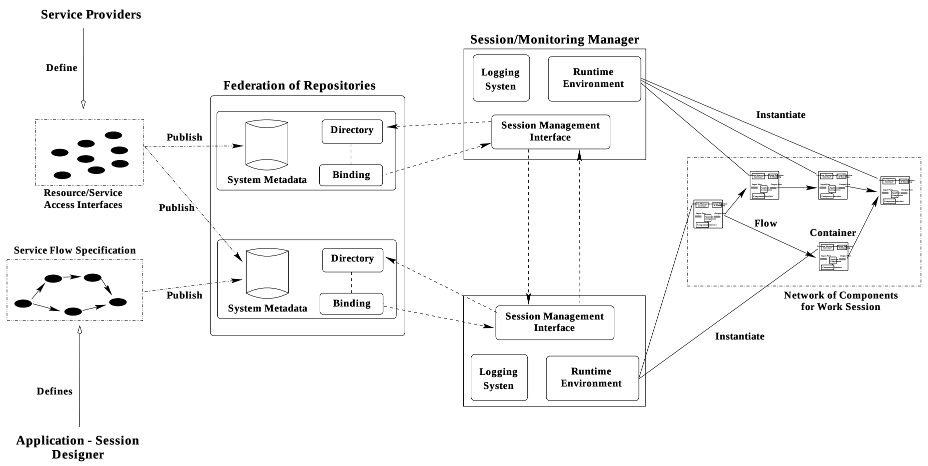

Service Providers
Define
Resource/Service Access Interfaces
Publish
Publish
Service Flow Specification
Publish
Defines
Application - Session Designer
Federation of Repositories
System Metadata
Directory
Binding
System Metadata
Directory
Binding
Session/Monitoring Manager
Logging Systen
Runtime Environment
Session Management Interface
Session Management Interface
Logging Systen
Runtime Environment
Instantiate
Flow
Container
Network of Components for Work Session
Instantiate

Figure 2.2: Overview of *Aurora* Architecture.

2.7 Composition via Scripting

In *Aurora*, the developer's task is to identify appropriate components and "plug" them together, via a form of *scripting*. A *script* is a set of software components with compatible input and output ports connected together to allow interoperation. The scripting model consists of the types of ports defined and the rules that determine *plug–compatibility* [NTdS91]. In the *Aurora* infrastructure, a connection between an input port and and an output port represent service availability, in the sense that it binds a client to a service provider. The execution model supports *peer–to–peer* interactions between *loosely coupled* participants, covering both coordination and collaboration. In *Aurora*, we achieve plug–compatibility by encapsulating services offered by resources in software component *containers* which export a uniform management interface. We synthesize on–demand composite services by linking the appropriate input and output communication ports, and thus implement a composite service as a network of cooperating components, in a run–time environment provided by a session manager and supervised by a monitor. A *container* is an extensible shell combining application–specific code that implements a step in a work session with an application–independent management interface, which enables *external control* of containers in a manner that is independent of the work session context.

Work sessions are implemented by establishing a network of active components, hosted by containers. Containers export a uniform interface for task/component management, which enables them to participate in work sessions supervised by a session manager. While a task (encapsulated in a container) remains active, it can retrieve data from its input port and forward data to its output port. Input and output are in the form of *streams*, which represent a flow of data as well as operation (action) invocations from a producer task to one or more consumer tasks. Each component is also expected to provide some form of instrumentation for monitoring and control purposes. The *Aurora* execution model assumes that application components encapsulated in containers export for these purposes part of their state, in the form of a list of attribute–value pairs. The uniform management interface provides methods for initializing, enabling, and disabling the instrumentation within an application component encapsulated in a container, and methods for retrieving and modifying the values of state variables exposed by the application component.

We have designed a scripting language, HERMES [MPN98c], to express scripts for work sessions, assuming that the resources of autonomous distributed service providers are accessible through software components that export certain uniform interfaces. The interpreter for the scripting language acts as a *mediator* between components. Such components are reusable because they are relatively context–free, in the sense that they have only a limited and well–documented set of dependencies on the run–time environment that hosts them and make no particular assumptions about other components that may invoke their services and interact with them. These aspects of components are discussed in Section 4.1. The application–specific configuration of components and in particular their interconnection in an ensemble is achieved by submitting scripts for execution to the work session manager provided by the *Aurora* infrastructure. An important aspect is that scripts themselves can be encapsulated in components, allowing the development of more complex scripts that combine components and scripts in higher–level software abstractions that appropriate for applications. Since the scripting model does not depend fundamentally on the programming language used for implementing components, script developers do not need to be fully aware of the details of the component infrastructure, as long as they understand the scripting model and select appropriate components for compositions. Visual development tools are well–suited for supporting this type of programming tasks.

Control and data flow are driven by events that are a combination of service request messages, state transition signals associated with tasks, as well as system–generated or application–specific notifications. Event–Condition–Action (ECA) rules [DHL90] specify the events and conditions that activate a task. The course of a workflow may change dynamically, as the next actions to execute are determined by rule triggering. Data and operation invocations flow among components through streams which are established as a result of rule triggering. Optionally, a component can provide additional state information by producing application–specific events. Thus, the results of automated tasks as well as the actions of humans may affect the workflow.

The *Aurora* infrastructure supports a *service flow* paradigm, where composite services are realized in the context of *work sessions* as flows of asynchronous service requests among components. Scripts describe the desired configuration of components for realizing a work process; however, this configuration can be inspected and manipulated at run–time. Dynamic configuration is achieved by invoking state inspection and control operations exported by the components encapsulated in containers. By allowing manipulation of components and their interconnections at run–time, workflow becomes more adaptive to the dynamics of the execution environment.

Each component that implements a task is required to notify its run–time environment when a significant state transition is to take place, particularly when the task begins execution (START), and when it terminates, either with success (DONE) or with failure (FAIL). Task execution may generate events for itself, as well as cause other tasks in a service flow to generate events. Optionally, a task can provide notification of the beginning and end of *phases* during its execution. Phases represent application–specific activity states and state transitions. Notification about the beginning and end of phases may be taken into account in the specification of work sessions. The *Aurora* run–time environment allows a task to be instructed to reset its original communication channels settings, and, furthermore, load dynamically a new application component. Thus, phase transitions allow run–time adaptation, which is essential for supporting dynamic and adaptive work sessions. Moreover, phase transitions allow the developer of an application component to expose task structure and state in detail, thus enabling monitoring of the task. An important aspect is that the instrumentation emebedded in components is separated from the definitions of rules that determine transaction boundaries, allowing external correlation of the events exposed by components. The developers of components only need to identify significant internal states and transitions, without having to embed transaction definitions in the instrumentation code. System administrators can independently define the transactions of interest and combine the notifications generated by the instrumentation emebedded in components to extract relevant measurements or to maintain audit trails.

2.8 A Note on Transactional Properties

Since we are considering open environments with multiple autonomous service providers, we cannot assume that all the providers involved in a session are able or even willing to participate in a distributed transaction under the supervision of a single (mutually trusted) transaction coordinator. Therefore, it is not possible to guarantee fully transactional semantics for the execution of HERMES scripts, as we cannot enforce desired semantics on work session participants. We only require that service providers provide notifications about significant state transactions for the requests that they execute. Such events can be combined using Boolean operators to express activation conditions for actions that need to be taken in order to handle partial failures. Provided that service providers export compensating actions to handle failures, it is possible to specify rules that emulate transaction commitment protocols (such as two−phase commit) or to handle partial failures in an application−specific manner. Therefore, rather than focusing on global consistency requirements, we provide mechanisms enabling *compensation* and *event tracking* via the *Aurora* monitor service.

Chapter 3

Related Work

In this chapter, we review related research and compare it with the *Aurora* approach presented in this dissertation in terms of objectives, approach, and scope. Section 3.1 focuses on related research on interoperability. Section 3.2 reviews research on distributed workflow enactment. Section 3.3 reviews research related to contract management, an area that has started to receive considerable attention. Finally, Section 3.4 focuses on research related to service−level management.

3.1 Interoperability

The InfoBus testbed, developed in the context of the Stanford Integrated Libraries Projects [PCGM^{+}96] (a project that was part of the DLI initiative), provided a comprehensive solution to the problem of interoperability in the domain of digital library services. The InfoBus testbed applies the OMG CORBA distributed object technology [COR94] by using wrapper objects to present a unified interface of digital library services, and a metadata architecture [BCGP97] to maintain metadata needed for interoperation of services. The component/container framework of the *Aurora* infrastructure serves as similar purpose, with specific provisions for interoperation of components that have been published in a directory service. The work session framework provides coordination support for general−purpose applications that fit the service flow paradigm, rather than more customized coordination for automating specific types of transactions.

An example of such a specific solution is given in [KGMP97] which presents event−driven models for different modes of consumer−to−merchant interaction, and an API to facilitate commerce transactions. Shopping models encapsulate the rules for specific types of commerce transactions and instruct the participants what to do next in the way of ordering, payment, and delivery. Another example is given in [RW97] which addresses the issue of rights management and discusses the automation of certain aspects of contract negotiation and enforcement. In comparison, the *Aurora* framework also offers a generic service−level agreement framework that can subsume the functionality described in [RW97].

The Infospheres project [CCD^{+}96] is concerned with theories for reasoning about concurrent composition of agents, and methods and tools for implementing sessions over the Web. [CCD^{+}96] describes a Java−based system that supports peer−to−peer communication among distributed processes. Each process is implemented as a multi−threaded object with a set of queues for incoming and outgoing messages. Such objects can be composed into sessions, by binding output queues to to one or more input queues. The focus of this project is on identifying and specifying software components that can be composed to create distributed applications, layered on top of standard network technologies such as the WWW. The proposed system model is similar to *Aurora*, as both approaches focus on applications requiring dynamic composition of services. There are however important differences in the overall aims and technical approach, as *Aurora* provides a run−time environment that can be monitored in detail for composing components that encapsulate services from autonomous providers, and, moreover, supports dynamic configuration and control of components and their interconnections. The Infospheres infrastructure does not include support services comparable in functionality with the *Aurora* container run−time environment, and does not support a service−level agreement abstraction.

WIDL [All97] is a metadata syntax, implemented in the XML meta−language [Bos97], that defines programmatic interfaces to Web content and services, so as to enable automated and structured access by client programs. WIDL definitions include the location (URL) of each service, input parameters to be submitted (via the GET and POST methods of the HTTP protocol), and output parameters to be returned by each service (as regions of returned documents). It is also possible to specify conditions for successful completion of a service and error indications to be returned to clients. Conditions further enable chaining of services, so that a client can issue requests that incorporate multiple services. Similar work is described in [Fuc96], which presents an approach in which domain−specific markup languages are used to handle interactions in a peer−to−peer system. These languages are understood both by software agents and the humans who interact with them. The objective is to dynamically integrate tools into distributed collaborative applications. SGML [Gol90] is used as the metagrammar system for specifying domain−specific markup languages.

[All97, Fuc96] address mostly the problem of describing and invoking a single service. In the *Aurora* infrastructure, descriptions of services (both CORBA−based and WWW−accessible) allow invocation of services with minimal knowledge of the implementation details of services. Clients can browse or search a directory service to identify target services, and then issue synchronous or asynchronous requests for obtaining results from these services. Moreover, services can be combined using the HERMES scripting language described in Section 4.5, either in a short−term basis for serving a particular request or in a long−term basis in the resource sharing context of a work session. Finally, the *Aurora* infrastructure explicitly addresses the issues of status tracking and on−line monitoring of services without compromising the autonomy of the service providers (as described in Section 4.1).

A compiler−assisted approach is taken in [BTJW98] that presents the *CHAIMS* language for composition of autonomously operated and maintained distributed components. This language provides primitives for preparing and issuing asynchronous service requests, checking their status, and extracting results. Wrappers hide the details of interaction with remote components, which may be complete computation and data servers. The language compiler generates client−side code for invoking the services exported by different components. By relying on statically−generated stubs at both the client− and the service−side, this system implicitly assumes that the participants in a workflow are willing to distribute and install stubs, which need to be kept up−to−date as the workflow evolves in time. The *CHAIMS* language does not support monitoring of service−level attributes.

The Programmer's Playground [GSM$^+$95] is a software library and run−time system that supports *I/O abstraction*, a high−level approach to interprocess communication. Relationships among distributed modules are established via connections among their published data structures, whereby updates of published data result in implicit communication according to the configuration of logical connections. Input is observed passively, or handled by reactive control within a module. This system targets mainly the requirements of collaborative multimedia applications, where autonomy and service−level management are not the essential concerns.

The HADAS system [BSCHL97] explicitly addresses autonomy issues. It provides an integration framework that hosts components, configuration support for establishing connections between components, and a scripting language (based on Scheme) for explicitly programming distributed computation involving these components. Connections effectively represent explicit agreements among autonomous providers for interaction of their components. Explicit links need to be established between sites in order to allow the use of remote components. The run−time system supports a protocol for automatic installation of component packages. The HADAS system does not support service−level agreements, and does not provide asynchronous request management and event notification services.

3.2 Distributed Workflow Enactment

Workflow management systems provided an automation framework for managing business processes, within and across organizations. Application domains where workflow technology is currently in use include telecommunications, health−care, manufacturing, finance and banking, and office automation. Workflow management systems provide support for the definition of business processes by composition of predefined tasks (processing steps). Workflow execution is based on the interpretation of process models. Representations of business models are maintained persistently in a database, and multiple instances of them can be instantiated at any given time. Workflow management systems allow organizations to re−engineer, streamline, automate, and track processes that involve both humans and information systems [SGJ$^+$96, KS95, GHS95]. Rather than having control and data flow embedded in application logic, steps in a workflow are performed according to the process model, with the workflow management system handling all issues of step invocation and data passing. Process models can be expressed in terms of work assignments for certain *roles*, rather than individual workers or resources, and a workflow management system can apply a *role resolution policy* to determine, from a pool of available and eligible workers or resources, the actual allocation at process execution time.

The view presented in this dissertation is that long−running processes spanning multiple organizations necessitate an infrastructure that supports asynchronous request management and reliable tracking of the actions of business partners, so as to support each partner in taking steps to handle failures and exceptions independently of all others while allowing co−operation under well−defined rules. In the *Aurora* infrastructure co−operation takes place in the context of work sessions that represent shared workspaces, i.e. dynamic collections of resources that can be utilized under the access restrictions imposed by their respective owners. The component/container run−time environment presented in this dissertation serves to minimize the dependencies among the co−operating parties, while providing mechanisms for issuing and tracking asynchronous requests for service performance.

Workflow technology is a broad research area that is concerned with modeling and automating processes that involve multiple tasks [GHS95, KS95]. However, most implementations so far have been centralized, with a single component responsible for sequencing process execution. There are few exceptions, and in the remainder of this section we proceed to examine them and compare them to our approach. A major aspect of our approach is that the *Aurora* application model assumes autonomous service providers, in contrast to implicitly assuming tight integration and a single authority in charge of management and administration.

Exotica/FMQM [AAA+95] and WIDE [CGS97] present distributed architectures for workflow enactment based on middleware technologies. The design of Exotica/FMQM is layered on top of a persistent queue system that supports transactional semantics for its queue access API calls. A process model definition is divided into parts by a compiler and each part is distributed to the appropriate node. Each node synchronizes with other nodes by communication over the network of persistent queues. WIDE is distributed CORBA−based architecture for workflow management, based on a database management system with active rule support. Each workflow engine maintains an event database, and includes a scheduler component that matches events to rules. Selected rules are recorded in a "to−execute" list, which is polled by a rule interpreter component. Both approaches appear to be more appropriate for the private networks of enterprises, since the dependence on specific middleware components imposes restrictions on the autonomy of service providers.

The INCA (*Information Carriers*) model is presented in [BMR96]. Work is carried out as interactions between an INCA, encapsulating an activity, and processing stations, which are assumed to be autonomous and may be only partially automated. The locus of computation migrates with the INCA, which includes the relevant execution context for the workflow, from one processing station to another. The INCA contains private data for the activity, which define the context shared by activity steps, a log of all actions (together their parameters) executed so far, and a set of rules defining the control and data flow among activity steps. Each processing station, upon receiving an INCA, performs the service requested and routes the INCA to the next destination(s). Rules encode control and data flow, as well as failure atomicity requirements. It is possible for a processing station to modify, add, or delete the rules and the data in an INCA. This agent−based approach provides considerable flexibility and autonomy, but seems more suitable for workflow involving mostly human workers. The set of rules embedded in an INCA could in principle encapsulate a service−level agreement, that would be fulfilled by executing actions at each processing station.

In the approach presented in this dissertation, a workflow may involve multiple service−level agreements, since agreements are are associated with specific services and made available by the service providers themselves. *Aurora* work sessions are established by creating networks of related components that co−operate to implement composite services. Co−operation is achieved by streams of operation invocations and data that flow between components, which are encapsulated in containers offering uniform interfaces for management purposes. This container framework, which enables detailed monitoring and control of active tasks and long−running compositions of tasks, is a major distinguishing feature of the approach presented in this dissertation.

The $METEOR_2$ project [SKM+96] has developed two prototypes, ORBWork based on CORBA and WebWork based purely on Web technologies, that support distributed workflow scheduling. The workflow specification is stored in a format which includes all the predecessor–successor dependencies of tasks and the definitions of data objects passed between tasks, as well as task invocation information. A code generator produces task management components and "wrappers", both for human and automated tasks. Each wrapper includes a hard–wired specification of the immediate predecessors and successors of the task that it manages, as well as code for evaluating the task's activation condition, invoking the task, and handling error recovery. After these components are installed manually at each site, users can instantiate processes. A centralized monitor provides limited support for tracking and monitoring. When activated, the task wrapper code gets the task input, by unpacking the data objects passed by the predecessors tasks, invokes the task code on the assigned processing entity, and, upon completion, determines the final state of the task, packs its output data objects, and signals the successors tasks.

In contrast, a task encapsulated in an *Aurora* container can relay operation invocations and data values to other tasks without restrictions, once the session manager has determined the component interconnections for the desired service flow. The run–time environment is more dynamic, as the specification of the component configuration for the session is interpreted, rather than pre–compiled and the network of components (hosted within containers) is dynamically established and can be modified at run–time. Furthermore, the actual resources to be used for each service step are dynamically determined, via the directory service.

PLEGMA [KZLO98] is an agent–based architecture for developing network–centric information processing systems. Originally derived from an environment of distributed image processing [Zik97], PLEGMA applies an auction–based mechanism for determining which processing node is to perform a particular task. Software agents represent the resources available in the run–time environment. Statistical profiles for service costs, including run–time estimates of data transfer time and execution time, are used in the selection process. Moreover, metadata describing the execution parameters of algorithms, or processing tasks in general, are maintained for use in the selection process, and in supporting sequences of task executions. PLEGMA has also been applied for managing workflow in the health–care domain. Overall, the PLEGMA architecture closely follows the WfMC reference model, but also provides functionality for dynamic resource allocation. It does not explicitly support asynchronous request management, and does not support service–level agreements as first–class objects. Quality of service requirements can be taken into account in the process of selecting a resource for executing a task, but there is no support for reliable tracking of conformance to agreed–upon terms.

[GSBC99] presents the *Collaboration Management Infrastructure* (CMI) project, which addresses the management of processes involving the integration of processes under the control of autonomous partners. The CMI project targets the requirements of applications such as crisis mitigation, command and control, logistics, and service provisioning in virtual enterprises. These applications are characterized by collaboration processes that require combined process and situation awareness, and cannot be effectively supported by existing workflow and groupware technologies. This aspect of the CMI project makes it particularly relevant to the research presented in this dissertation.

The service model of CMI enables the definition of abstract service interfaces that specify the application–specific semantics of services by describing their input/outpout parameters and state transition diagrams with application–specific states and operations. Such state transition diagrams capture the conversational interaction protocol between service providers and their customers. Conversational service coordination is solely conducted on the basis of the service interfaces. The CMI project investigates service brokering, the process of selecting among providers of services that conform to a given interface specification. The selection takes into account quality–of–service attributes.

In comparison, the *Aurora* infrastructure allows service providers to make commitments on service–level attributes of their services, and provides reliable means for such commitments to be monitored and enforced at run–time. This is a major distinguishing aspect of the *Aurora* infrastructure. Moreover, service providers can expose important state transitions of their published components, and thus allow any authorized external entity to monitor event notifications about these transitions. Moreover, the asynchronous request management primitives supported by the container run–time environment allow control of conversational services. It is important to note that service–level management, asynchronous request management, event notifications, and selective exposure of component state are offered as generic services for applications, as part of a component/container framework that builts upon the well–established OMG CORBA platform for distributed objects. Therefore, these services are generic, rather than being available only within the confines of a specific application run–time environment.

3.3 Service Contracts

The framework for service–level agreements presented in this dissertation allows the interactions between service providers and their customers to take the form of contract fulfillment, by providing to all the parties involved an explicit and durable means of documenting the mutually accepted terms of service, the steps taken towards implementing these terms, and the notification and compensation actions to be taken in the event of deviations from the expected behavior. Contracts provide a powerful framework for expressing and managing complex relationships between transaction participants. This has been illustrated in the context of rights management and access control [SL97, RW97], which are key issues as the interest in electronic commerce is growing [Sza97]. Contacts provide a powerful framework for expressing and managing complex relationships between transaction participants. The research presented in this dissertation extends the scope of such contracts to include terms related to expected performance and exception handling behavior. This extension is deemed necessary for enabling more predictable and manageable services in dynamic open environments.

[Lud98, FFH^{+}98] point out, using examples from the insurance service industry, the issues of customization and on–line control of the fulfillment processes for long–running complex services, through invocations of control operations and feedback loops. In a similar vein, [LW99] outlines an approach to cross–organization workflow that relies on dedicated gateways between the independent workflow systems that encapsulate the details of agreements for co–operation. [Lud99] discusses the problem of assigning the costs due to deviations from the expected process flow in an interorganizational settings. This functionality, falling under the more general category of on–line control of a long–running fulfillment process, is covered by the service–level agreement framework presented in this dissertation.

The *Coyote* system [DP97a, DP97b] supports the specification and enforcement of *business deals* among providers, based on code generation tools that generate server–side and client–side code for enforcing the terms of service contracts. The *Coyote* system supports the *service transactions* model, which includes service actions that implement activity steps, persistent conversations built from actions, scheduling rules that determine the "next" action in response to events, and a compensation paradigm to maintain integrity. The *Coyote* system assumes that user actions control the flow of execution between blocks of transaction code that implements steps in a complex service.

By relying on statically–generated stubs at both the client– and the service–side, this system implicitly assumes that the participants in a workflow are willing to distribute and install stubs, which need to be kept up–to–date as the workflow evolves in time. This is a fundamental limitation of approaches that rely on statically–generated stubs. In contrast, the approach taken in this dissertation is to provide a platform for hosting proxies to services and an asynchronous request management protocol to allow clients to issue, monitor, and control requests without having access to stubs specific to the target services. The platform relies exclusively on metadata published by the providers about their services, and the dynamic interface invocation mechanism of CORBA [COR94].

An IETF Internet Draft [Swe98] describes the *Simple Workflow Access Protocol*, that allows a client to initiate, control and monitor asynchronous long–duration service execution at remote sites. Related work is also reported in [PPC97b] that presents an architecture for supporting workflows involving autonomous participants and service providers, based on asynchronous requests among workflow participants and standard interfaces for *sources* and *performers* of work items. In our approach, we share the emphasis on defining a generic asynchronous request management infrastructure, and complement that with a flexible work session framework and service–level agreements as first–class objects.

Extended and flexible transaction models [Elm92] place particular emphasis on the specification of tasks and their outcomes. However, by focusing on the low–level of detail of individual data manipulation operations, transactions are classified only as successful or failed, thus restricting the ability to reason about transactions from a contractual point of view that can capture failure and exception handling as part of handling contract violations.

The APRICOTS prototype [Sch93] implements the ConTract extended transaction model [WR92]. This approach supports long–lived transactions by allowing the developer to commit results as early as possible in the course of a transaction, while having the option to define compensating actions that will be executed in the event of failures to restore an acceptable global system state.

[VP98] presents an object framework that uses interacting, possibly nested transactions in implementing the fulfillment phase of contracts that describe obligations related to the delivery of business services. In this framework, contracts are explicitly represented as objects that mediate the execution of transactions.

3.4 Service Level Management

Monitoring is an essential issue in managing interactions that combine the services of autonomous service providers. This dissertation shares the view expressed in [Wei95], which argues that "workflow monitoring is an important link between the computer–science and the business–science perspective of workflow management". The logging and monitoring infrastructure presented in this dissertation corresponds to the audit trail approach of [Wei95] for providing an *application–level log*, with a facility for correlations and aggregations.

Service level management for workflow in open systems has not been investigated in–depth, although it provides a link between issues related to workflow infrastructure and actual business goals. As argued in [KTL^{+}92] (in the context of software process management), establishing performance baselines, setting improvement goals, explicitly defining processes, and measuring progress toward goals are essential aspects of process management.

Service Level Management is an activity that has long been practiced in enterprise data processing centers, typically on mainframe computing systems, which provide comprehensive monitoring and resource control facilities. IBM's Workload Manager (WLM) for the MVS/ESA [BE95] allows an installation to group units of work in "service classes" and define *performance goals* for them. Thus, installation managers explicitly state to the operating system the service performance goals towards which the units of work should be managed. The relative priorities of classes of units of work are reflected in the allocation of available resources to units of work, using dynamic resource allocation policies [NFC92, FNGD92, FNGD93] to dynamically adjust access to processor and storage resources.

This is not yet the case with open systems, primarily due to the lack of a standard way to monitor management metrics and enforce resource access controls; however, there are some recent developments towards service level management for client/server business systems [McB96]. An important development towards this direction has been the introduction of the Universal Measurement Architecture [UMA97] and the Application Response Measurement (ARM) API [CC96]. UMA provides services focused on controlling the acquisition, delivery, and management of performance–related data and events in distributed multi–vendor client/server systems. Transactions in distributed computing environments are not fully contained in a single system, as in the case of centralized systems. Therefore, they are difficult to track across systems. Tracking requires both tagging of performance–related data about the components of a unit of work and a mechanism for gathering and correlating this data from multiple sources. Both of these requirements are outside the scope of UMA. The ARM API is designed to cover the first of these requirements, enabling instrumentation of applications so as to delimit "transactions" of interest, by inserting `start, update, stop` indicators around sections of application code. The design of the *Aurora* monitoring infrastructure subsumes the functionality of the ARM API, since application components can produce log records delimiting transactions of interest, and, furthermore, supports *correlation* and other types of navigational queries over the interaction history of complex units of work, as a *generic* service for applications.

[Kic96] argues that it is beneficial from a performance point of view for software module developers to expose implementation details to clients. This is a form of *computational reflection*, supported by offering two interfaces. The primary interface provides the module's functionality, while the *meta–interface* allows clients to adjust certain implementation options that underlie the primary interface. The proposed SLA abstraction allows service providers to selectively expose implementation details to clients, thus enabling monitoring performance metrics and control of operational parameters. The actual implementation of monitoring and control is by instrumentation provided by service component developers. The SLA abstraction hides such details, providing a uniform interface for clients.

[Gun95b, Gun95a] propose extending the methodology of benchmarking beyond the current paradigm of treating the *system under test* as a "black box" to which a predefined "stimulus" (such a SPEC or TPC benchmark workload) is applied and performance statistics are collected and aggregated into the *external* performance metrics specified by the benchmark standard definition. The drawback of such benchmarking is it is not straightforward to explain *how* the reported metrics were attained. Analysts can only make conjectures based on metrics. A "flight recorder" paradigm is proposed, that involves performing *internal* measurements as well, allowing performance analysts to track the "flight" of a unit of work through the system under test, as the unit of work consumes resources by passing through a chain of components. This requires installing measurement probes in the components involved, which may span several client and server systems in the case of complex units of work, and it is argued that UMA can provide a standardized infrastructure for the capture and transport of distributed performance data. In this paradigm, a benchmark specification includes the definition of the instrumentation points that must be measured and reported. The approach proposed in this paper not only adopts the performance measurement paradigm of [Gun95b, Gun95a] but, additionally, addresses the issue of multiple autonomous service providers. Each provider publishes a SLA for its clients, but there is no single authority for supervising all the providers involved in a complex unit of work in order to maintain the integrity, performance, or security constraints promised by SLAs. Therefore, each service provider can only enforce pair–wise SLAs with its clients, and has its own *partial* view of the interactions involved in a complex unit of work.

[CD97] presents a simple language that allows the expression of common strategies for handling failure and slow communication when fetching content on the Web. In this context, a service is an HTTP–accessible content provider wrapped in error detection and handling code. The language provides *service combinators*, which are operators for composing services taking into account both their output and their failures. The semantics of the language are defined in terms of the effects of these operators on the state of services. The service–level agreement abstraction presented in this dissertation provides the basis for automated interactions, including handling failure and slow communication, as it captures the expected behavior of service providers and can trigger the execution of automated actions.

Interactive steering is the "on–line configuration of a program by algorithms or human users, where the purpose of such configuration is to affect the program's execution behavior" [SES+97]. Falcon [GEK+94] is a set of tools and libraries supporting on–line monitoring of application–level events. Falcon provides a sensor specification language, a compiler for generating application sensors, and uses one or more local agents for on–line information capture, collection, filtering, and analysis. Monitoring/steering middleware is provided to disseminate monitoring events to the clients that wish to inspect and interact with the running application. The proposed SLA abstraction can provide the basis for on–line monitoring and steering. Futhermore, the *Aurora* monitor supports correlation of log records related to a complex unit of work, that may involve multiple service providers and participants.

Chapter 4

Component/Container Framework

In our approach, in the context of the *Aurora* project, we take the view that dynamic workflow is best supported by a distributed component infrastructure providing standard services for locating potential workflow participants and managing service requests. By standardizing these aspects of workflow execution, service providers can participate in distributed workflow processes without having to agree upon a common workflow specification language or to maintain accounts and work−lists on multiple workflow management systems (see also [PPC97b]). Participants remain autonomous, and can be held responsible only for work that is performed within their own scope of control. Workflow, involving scheduling and coordinating work among multiple service providers, management of control/data flow and exception handling, is reduced to issuing requests to service providers, as no global authority has overall control over the service providers.

4.1 Component Model

We have implemented a CORBA−based component model as the basis of our work session infrastructure. Aspects of this run−time infrastructure have been previously presented in [MPN98b, MPN98c, MPP98, MPN99]. Table 4.1 summarizes the structure of a component according to the *Aurora* model. The following subsections describe the functionality offered through these interfaces. The component model was first presented in [MPN99].

Management Interface	Functionality
`Exporter`	dynamic collection of exported service object references
`Importer`	dynamic collection of imported service object references
`Reactor`	receiver of event notifications
`Notifier`	notifier of events to registered subscribers
`Configurator`	collection of configuration properties
`ComponentControl`	collection of exposed state variables and control operations
`ComponentBase`	“front−end” for component

Table 4.1: Management Interfaces Exported by Aurora Components.

4.1.1 Basic Concepts of the Component Model

A component provides one or more functional interfaces and may use interfaces provided by other components or infrastructure−provided services. A component may also emit notifications for events related to its internal operation, and monitor event notifications generated by other components. By defining interface connections and event propagation/consumption relationships among components, developers establish *communication channels* among components. The *Aurora* run−time environment allows inspection and manipulation of a component's channel settings at run−time. Moreover, each component has configurable properties, some of which can only be set at instantiation time while others can be manipulated at run−time as well. Configuration and other monitoring and control operations, such as retrieving the values of certain state variables and setting parameters that affect performance, are enabled through a `ComponentControl` object to be implemented (optionally) by the component developer.

The instantiation of a component in the run−time environment of a service provider requires references to the CORBA objects that implement its exported service interfaces, and references to any external services that they depend upon. The former are maintained by the `Exporter` object, while the latter are maintained by the `Importer` object. Instantiation of a component may also require that the component subscribes to event notifications from various sources, including other components. Subscription entails that the component's `Reactor` object is instructed to receive notification from the appropriate sources, and a corresponding callback object is specified for each source. This arrangement allows the component to react upon asynchronous notifications of events that affect its operation. Finally, component instantiation requires that the component's configuration properties must be set accordingly, via the `Configurator` interface. Examples of configuration properties include settings for connecting to database systems and other data sources, and threshold values for various performance tuning parameters that affect the internal operation of the component. By invoking a method `finalize_configuration()` on the `Configurator` object, the component becomes ready to accept and service customer requests.

4.1.2 Management of Exported Interface References

A component provides one or more services, through its `Exporter` object that maintains a dynamic collection of CORBA references to objects that implement the services. A customer can browse through this collection or select a service by name. Once a selection is made, the customer can issue a request to the selected service using the standard invocation mechanisms of CORBA. A service provider is free to add or remove object references to this collection, thus being able to introduce, withdraw or upgrade services without having to shutdown and restart the component.

4.1.3 Management of Required Interface References

The object references maintained by the `Importer` object represent services required for implementing the component's functionality, and may include services offered by other components as well as support services offered by the container run−time environment. It is possible for a component to maintain multiple (redundant) references for the same service, allowing fail−over in the event of failure to contact a service.

4.1.4 Management of Event Notifications

When activated a component may generate events, such as significant internal state transitions or advisory notifications, that may be needed by other components. The `Notifier` object of a component maintains dynamic collections of the components that have registered to receive notifications of events from this component, and allow the source component's implementation to forward notifications when appropriate.

4.1.5 Component Metadata

Components are described by metadata, maintained by a directory service that complements the functionality of CORBA's Interface Repository by describing the interfaces provided and required by a component, the events that it can emit and monitor, its access control restrictions and supported service–level attributes, its exposed state variables and control operations, and its configuration options. Component descriptors are expressed in the form of sets of attribute/value pairs, which can be nested and thus form a tree that can be navigated to extract portions of interest to a customer. The directory service also allows providers of components to define *trading properties* encoded as lists of name/value pairs, to support clients in selecting among similar components.

4.1.6 Component Installation and Support Services

A component ultimately takes the form of a *package* consisting of files that contain the component's implementation as well as metadata describing the component (such as an access control list, estimates of expected performance, and supported compensating actions for exceptions). A component is deployed by installing a package in the run–time environment provided by a container, which provides components with access to several support services and maps method invocation requests to threads from a shared pool. A component package can be pre–installed in the run–time environment of a container, or can be obtained via a service of type `ComponentClassProvider` that operates under the control of a service provider or a trusted third–party.

The support services offered by the container consist of a CORBA run–time environment including an Object Request broker and instances of the OMG COS Name Service and the OMG Interface Repository, a directory of component descriptors, a generic component loader and factory service, services for volatile and persistent data storage, and several variants of services for asynchronous notifications. The container also offers a scheduling service for automatic execution of action sequences when a condition, specified as a boolean condition over monitored event notifications, becomes true (`ECA` scheduler), as well as a time–based action scheduling service (`TIME` scheduler) that supports the execution of actions at a specific point in time, or periodically. The `TIME` scheduler also supports periodic invocations of *probe* operations on objects that implement the `ProbeMonitor` interface. This interface allows periodic checks on the current state of running services. As a specific example, a `ProbeMonitor` implementation has been developed to measure the response time for fetching the first page of WWW services, such as the Yahoo directory and the Altavista search engine, and characterizing the current state of the service as one of `responsive`, `slow`, or `inaccessible`, based on configurable thresholds.

Finally, a specialized class of containers, of type `WorkRequestManager`, offers an asynchronous request management service, described in detail in Section 4.2, and a persistent logging system that provides a request monitor service with support for aggregation and correlation of request–related log records, described in Section 5.2.

4.1.7 The ComponentProxy Interface

For a container to be able to mediate accesses to services provided by a component, the component developer must provide a software module that implements the `ComponentProxy` interface. This interface merely provides an identification of the component package, and one or more object references to instances of the component's `ComponentBase` management interface, which in turn provides references to the component's standard management interfaces as well as its functional interfaces. This reference is not exposed by the container to customers, thus prohibiting direct interaction with the service and preserving the autonomy of the provider. In this way, the container mediates all interactions of customers with the service implementation, after performing access control checks and logging, and allows the customer to interact with the target service only within the constraints defined by the service provider, which makes available the package containing the `ComponentProxy` object.

A component developer may provide an implementation of the `UpdatableComponentProxy`, a specialization of `ComponentProxy` that allows a service provider, or a third–party service that monitors the implementation of the service represented by the proxy object, to maintain the reference encapsulated in the proxy up–to–date. When there is a change on the CORBA object reference for the corresponding service, or when a reference has become invalid, or even when alternative references for a given service have become available, the service provider that owns the service notifies all containers that are known to host instances of proxy objects for the specific service. The container, in turn, invokes call–back operations exposed by the `UpdatableComponentProxy` interface to reflect the changes signaled by the service provider, with minimal impact on its customers. Customers are to a large extend shielded from such changes, since they do not directly use object references for services, but rather rely on the container to mediate their requests. Customers need only to preserve the persistent request identifiers returned by the container in response to their requests (for example by using the persistent logging service provided by the *Aurora* infrastructure). Notifications of changes regarding the actual object references for service implementations provide a crucial building block for robust and scalable operation in the face of long–duration workflow processes. Support for alternative references for a service allow the container to coordinate load sharing and fail–over without the explicit cooperation of customers. Such functionality can be requested by setting appropriate attributes in the service–level specifications that accompany requests issued by customers.

An additional measure towards scalability is that a container can "swap–out" a component instance, provided that it supports the `PoolObject` interface, which is a specialization of the `UpdatableComponentProxy` interface. This interface standardizes the methods that the container must invoke in order to inform the component instance that it has been selected for eviction from the space of active component instances, and give it a chance to preserve its state, as well as the methods for restoring previously stored state. The current prototype uses a simple least–recently–used (LRU) policy for selecting the component instance to be evicted when a container–specific threshold on the number of active component instances has been exceeded. Thus, there is an upper bound on the number of concurrently active proxy objects that are hosted by the container, and, moreover, reuse of proxy objects is possible. When a customer invokes a request management operation with an evicted component instance as the target, the container uses the `PoolInterface` to restore the original state. This functionality relies on the persistent storage service provided by the *Aurora* infrastructure.

4.1.8 Support for On–line Monitoring and Control of Components

Optionally, a `ComponentProxy` can provide a reference to a `ComponentControl` object, that is part of the actual service implementation at the service provider's site and exports state variables and control operations. This functionality requires instrumentation embedded within the service implementation. The reference for the `ComponentControl` object is not directly exposed to the customer; rather, the container delegates monitoring and control operations to the `ComponentControl` object upon request by customers. The `UpdatableComponentProxy` provides operations for keeping the reference to `ComponentControl` objects up–to–date. Table 4.1.8 summarizes the methods of the

Method	Functionality
`Start`	(re)start with given parameters
`Stop`	shutdown execution
`Suspend`	temporarily suspend execution
`ListStateVars`	metadata describing exposed state variables
`ListControlOps`	metadata describing supported controls
`RetrieveState`	retrieval of specified state variables
`SetState`	update of specified state variables
`InitInstrumentation`	initialize internal instrumentation
`EnableInstrumentation`	enable (selective) inspection/updates
`DisableInstrumentation`	disable (selective) state inspection/updates
`StopInstrumentation`	cancel internal instrumentation

Table 4.2: Methods of the `ComponentControl` interface.

`ComponentControl` interface that allow a customer to perform on–line monitoring and control operations on a component, provided that the component's implementation, which is the responsibility of the component's owner service provider, supports these operations. If not supported, a method invocation on the `ComponentControl` interface should raise an exception of type `UnsupportedControlOperation` so as to notify the caller that the requested method is not implemented. This design convention allows the `ComponentControl` interface to remain uniform across components that contain instrumentation code of widely varying sophistication. The uniformity of the `ComponentControl` interface is essential for managing ensembles of components that have been configured to co–operate in the context of work sessions.

4.2 Management of Asynchronous Requests

Figure 4.1 illustrates the structure of a container. The container exports a standard interface for asynchronous request management, and handles the low–level details of monitoring their progress and taking prescribed actions upon their completion or upon exceptions. These actions include callback invocations on customers, initiation of alternative actions, and fork/join primitives for concurrent request scheduling. This request management interface is used in the implementation of work sessions and service

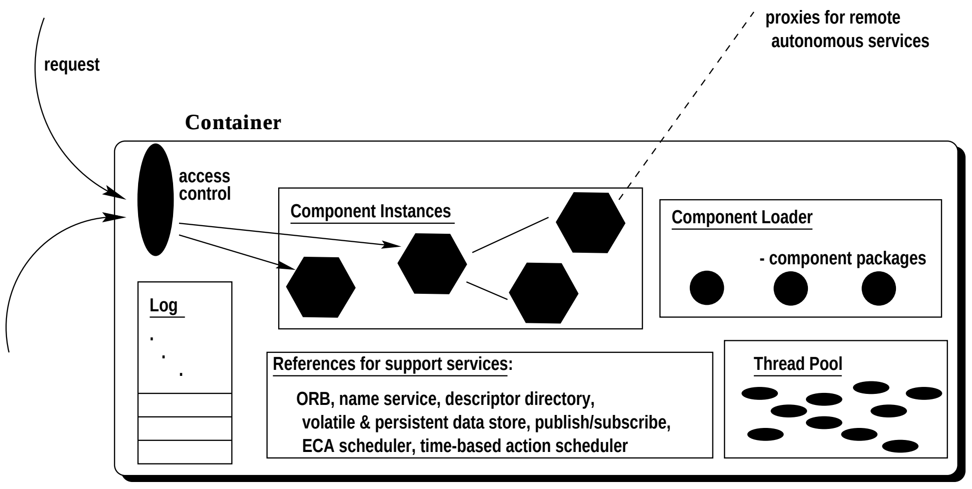

proxies for remote autonomous services
request
Container
access control
Component Instances
Component Loader
- component packages
Log
References for support services:
ORB, name service, descriptor directory, volatile & persistent data store, publish/subscribe, ECA scheduler, time-based action scheduler
Thread Pool

Figure 4.1: Structure of a container hosting ComponentProxy objects.

level agreements. Figure 4.2 presents an state transition diagram describing the life−cycle of a request.

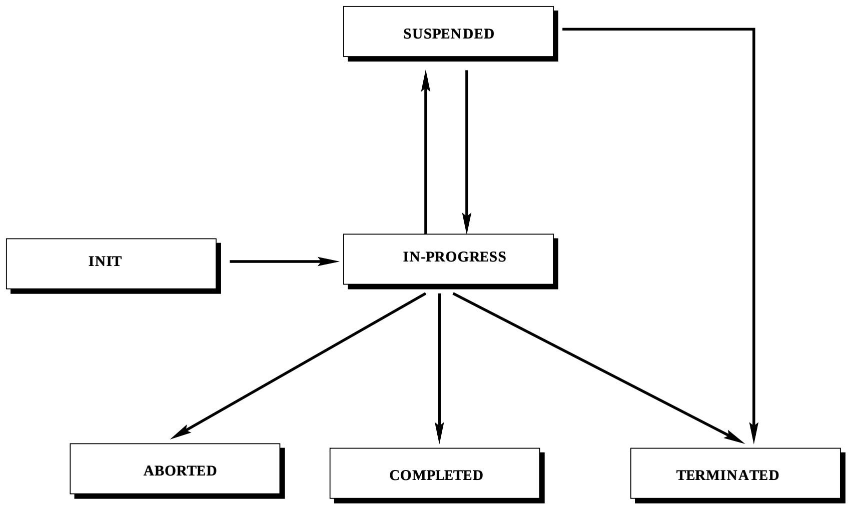


Figure 4.2: States in the processing of a work request.

4.2.1 Asynchronous Requests in CORBA

Applications that perform a series of tasks that must be done sequentially cannot benefit from asynchronous communication. Applications that make only short duration remote operations have little need for asynchronous communication.

Asynchronous communication can allow an application to perform additional tasks instead of waiting for tasks to complete. Applications that have a number of tasks that can be performed in any order can often benefit from distributed asynchronous communication. This becomes more important for applications that call lengthy remote operations. In order to benefit from asynchronous communication, an application must be able to perform some task after the request is issued but before the response is available. Tasks might include prompting for additional user input, displaying information, or making additional remote operation requests. Typical asynchronous communication candidates include applications that need to perform several lengthy database queries or complex calculations.

At the lowest level CORBA supports two modes of communication:

- synchronous request/response: allows an application to make a request to some CORBA object and then wait for a response.
- deferred synchronous request/response: allows an application to make a request to some CORBA object. An empty result will be returned immediately to the application. It can then perform other operations and later poll the CORBA Object Request Broker to check if the result has been made available.

At the lowest level, the CORBA deferred synchronous communication does allow a certain degree of asynchronous communication. Polling for responses represents only one form of asynchronous communication. Other more sophisticated asynchronous communication can only be achieved by developing mechanisms on top of the lowest levels of CORBA. The *Aurora* component platform presented in this dissertation provides such mechanisms, as described in Section 4.2.2.

Moreover, while CORBA does support a deferred synchronous request/response, it does not directly support distributed requests with a callback−driven response. A callback−driven response allows an application to perform an operation on a distributed object, associate a callback with the response, continue with other processing. When the server responds, the associated callback is automatically executed within the original caller's application. Callback invocation also enables a client or a third−party monitoring service to track the current status of a request, and receive up−to−date information about the request's progress (including partial results). Section 4.2.3 describes the callback−driven response mechanisms provided by the *Aurora* component platform.

4.2.2 Primitives for Asynchronous Request Management

Table 4.2.2 presents the primitives provided by the Aurora container for managing asynchronous requests. Requests are represented by persistent identifiers, assigned by the container, of the following form:

$$request://<cref>/<cK>/<iK>/<seq\#>?<pspec>.$$

A request identifier includes a reference $cref$ to the container that handles the request on behalf of the client, an identifier cK specifying the class of the target component, an identifier (key) iK for the specific component class instance, a container−generated sequence number for the request, and a specification of parameters $pspec$. The parameters specification is of the form

$$c=<caller>: i=<interface>: m=<mX>: sla=<slaID>,$$

thereby providing an identification for the client that issued the request and specifying the target component's interface to be used for performing the requested service (as specified by the `method` parameter). The `sla` parameter specifies which of the service–level agreements supported by the component is to be applied in processing the request. The actual request parameters are recorded in the persistent log maintained by the container when the request is issued. Similarly the results (including partial results that a service may provide to the client as they become available) are logged by the container, thus providing a complete audit trail for the execution of the request and assisting the implementation of compensating actions to handle exceptions. The asynchronous request management

Method	Effect
`request_init`	prepare request for submission to service
`request_start`	submit a prepared request
`request_issue`	combination of `request_init` and `request_start`
`request_cancel`	attempt to cancel a submitted request
`request_results`	retrieve (partial) results of request
`request_wait`	block until (final) results of request become available
`request_fork`	submit concurrently a set of prepared requests
`request_join`	block until at least the specified number of requests from a set complete
`request_with_compensation`	submit prepared request and specify compensation request
`request_with_time_limit`	submit prepared request and specify maximum acceptable delay
`list_pending_requests_limit`	identifiers of requests that have not yet been processed

Table 4.3: Asynchronous request management primitives.

primitives were first presented in [MPP98].

It is important to note that the request management primitives are exported by *the container*, rather than by the components hosted in the run–time environment provided by the container. The container handles all the details of supporting asynchronous request management for services that do not necessarily support asynchronous interaction. Moreover, the container does not expose the actual references to the components that implement services. This design decision introduces a level of indirection in the interaction of customers with service providers, a necessary step for enforcing access controls and reliable tracking of requests in the context of service–level agreements.

The container supports asynchronous requests by using multiple threads of execution to handle concurrent requests. Multi–threading supports concurrent processing within a particular server that hosts the container. An application that needs to perform concurrent distributed requests can issue requests in different threads, without having to handle the details of thread synchronization and management, by using the request management primitives exported by the container. The container incorporates a pool of re–usable threads that can be assigned to perform the processing for a particular request. Upon successful termination of processing the request, or upon detection of a run–time exception or system failure during the processing of the request, the container returns the thread that handled the request to the pool of available threads. By maintaining a pool of threads, rather than creating a new thread for each request, the container reduces the overhead cost of processing an asynchronous request.

The actual processing of a request is performed by having a thread issue a request via the Dynamic Invocation Interface (DII) supported by CORBA [COR94]. This method of issuing requests has the advantage that the container does not need to have stub code for the target service. The `ComponentProxy` object that is hosted in the run–time environment provided by the container only needs to offer a reference to the target service, and this reference is not exposed to the clients of the service. This reference may be an Interoperable Object Reference (IOR) [COR94] for a CORBA object that directly implements the target service, or an IOR for an *Aurora* component that exports the specified interface. In the latter case, the container must obtain a reference to the component's `Exporter` interface, and use that interface to obtain a reference (IOR) for the specified service.

The metadata published by the service provider in the directory service offered by the *Aurora* component platform allows the container to issue a request without having access to stub code generated by an IDL compiler. The parameters of the primitives `request_init` and `request_issue` suffice for the container to extract from the target service's metadata the method name and the identifier of the interface of the target service, as declared in the Interface Repository service (IFR) specified by the OMG CORBA standards [COR94]. An IFR service is part of the run−time environment provided by a container. The IFR identifier allows the container to obtain a machine−readable description of the target service's interface. This description is used by the container to perform marshaling of the supplied request parameters, and to extract the results of the method invocation. These tasks would be conventionally handled by stub code as generated by an IDL compiler; however, by relying on stubs an implicit coupling between the service provider and the container (and in turn its customers) would be enforced, as the service provider would have to handle the task of distributing appropriate stubs to all its clients. By using the DII invocation mechanism, the service provider only has to publish (and maintain up−to−date) the metadata describing its services. This arrangement allows for fully autonomous service providers, by minimizing the degree of coupling between service providers and their clients.

4.2.3 Callbacks for Notifications about Asynchronous Requests

CORBA communication is inherently asymmetric: Request messages originate from clients and responses originate from servers. It is important to realize that a CORBA server is a CORBA object, while a CORBA client is really a CORBA stub. A client application may use object references to request remote service, but it may also instantiate CORBA objects and thus be capable of servicing incoming requests. Along the same lines, a server process that implements CORBA objects may have several object references that it uses to make requests to other CORBA objects. Those CORBA objects may reside in client applications. By implementing a CORBA object within an client application, any process that obtains its object reference can notify it by performing an operation on the client−located object. This approach is applied in the design of a callback mechanism for asynchronous requests in the *Aurora* component platform.

Table 4.2.3 presents the callbacks that a container can invoke on an object of type `CallerCB` provided by a client that prepares a request and issues it to a target service using the container's request management primitives (presented in Table 4.2.2). Figure 4.3 illustrates how a customer interacts with services provided by a component that is hosted within a container.

Method	Effect
`request_ready_to_start`	notification after `request_init` that the processing of a request can be started
`request_accepted`	confirmation that a submitted request can be processed
`request_refused`	explicit declaration that a submitted request cannot be processed
`request_in_progress`	*heartbeat* from target service, with partial results
`request_completed`	notification of successful completion, with final results
`request_aborted`	service failure notification
`request_delegated`	notification of request delegation

Table 4.4: Asynchronous request management callbacks.

4.3 Access Control Framework

Access control in the *Aurora* architecture is based on the notions of actions, actors, roles, and access controllers.

Actions include using an interface exported by a component, invoking a specific method on a target component that represents a service, and monitoring event notifications emitted by a target component. Actions are initiated through method invocations, which are intercepted by the container. Actions related to using interfaces and monitoring event notifications are used for establish a network of cooperating components. Such component assemblies implement distributed multi−party work sessions.

Actors are entities that initiate actions in the course of a work session, identified by security credentials. An actor may have a set of descriptive properties, such as contact information for a human involved in the work project represented by the session and machine−readable specifications of the capabilities of a computational service. Unique security credentials are associated with each session participant, through directories made available by *certification authorities*, and are passed along with each request, as part of the request context. Credentials include security−related information about an actor, including its identity, public−key certificate, and certification authority.

Roles represent sets of actions that an actor may initiate. A Role object may be shared by multiple components. *Access controllers* define the permissions/restrictions for actors participating in a work session to assume given roles, thus determining whether a particular requester can initiate the actions associated with a role, on components that support this role. Session participants wishing to execute an action on a component have to be explicitly allowed to assume one of the *roles* specified by the authority that is responsible for the component. The object interaction diagram in Figure 4.4 illustrates the flow of method invocations for establishing whether a requester can initiate an action in the context of a work session that involves components hosted in containers that intercept all incoming requests.

It is possible to have multiple access controllers associated with the same component, depending on the application scenario. Multiple roles may share the same access controller object, thus sharing the same access control policy. Actions, roles, and access controllers are expected to be defined independently by the developers and managers of autonomous services. Whereas actions are tied to specific target objects, roles as well as access controllers may be developed by independent authorities other than the service provider, for the purposes of enforcing a customized access policy for a *composite service* as enabled by the collection of resources available in the sharing context of a work session (see Chapter 6).

4.4 Support for Event–Driven Execution

The *Aurora* infrastructure provides several services for asynchronous notifications. References to these services are made available through the container to components. This allows for example developers of Task components (described in the Section 6.1) to publish notifications about state transitions and significant events during their operation to registered subscribers. Subscribers receive a `Channel` object that allows them to poll for pending notifications, block until a notification is received, or register a callback to be executed automatically upon arrival of a notification. All variants of the publish/subscribe services offered by the *Aurora* infrastructure provide their customers with `Channel` objects, thus simplifying the work of component developers. The variants are differentiated by the level of detail that they allow in the specification of events of interest. What is more, `Channel` objects are returned also by the ECA and TIME scheduler services provided by the infrastructure. This uniformity allows for composition of the mechanisms for event–driven execution, thus enabling the implementation of complex sequencing rules. This functionality is utilized by the scripting language *HERMES*.

The basic publish/subscribe service provides `SubjectGroup` objects for allowing components to handle the details of keeping track of subscribers that receive notifications for all events published by the component on the *subject* represented by the *SubjectGroup* object. Event notifications are objects that provide a string tag identifying the event and a list of attribute/value pairs for carrying component–specific state. The receivers of event notifications are expected to be able to extract this state information, which is opaque to the infrastructure services.

`SubjectGroupFilter` is a specialization of `SubjectGroup` that allows a customer to request notifications only for events published on a subject that satisfy a boolean predicate expressed over the state included in the event. The predicate is encapsulated in an object of type *Filter* that is provided by the customer to the `SubjectGroupFilter`. *Filter* objects are constructed using a factory service, which is provided by the infrastructure, based on textual specification of the predicate. When an event source publish an event on a `SubjectGroupFilter`, the *Filter* objects of the subscribers are invoked to determine whether the event should be relayed. Although the infrastructure does not comprehend the semantics of this state information, which is application context–specific, it can mechanically invoke the predicate test method of *Filter* objects.

Another specialization of `SubjectGroup` allows for notifications based on a finite state machine model. An object of type `SubjectGroupFSM` encapsulates a definition of a finite state machine, in the form of a graph where nodes represent states and arcs represent state transitions that take place when specified events have been signaled. The specification allows the distinction between *final* and *non–final* states. A customer can subscribe to receive notifications when either a specific state is reached or when a specific transition takes place. For example, for a simple workflow specification that can be described by a finite state machine specification, a customer can register a callback to be invoked when the workflow reaches one of its final states for monitoring purposes or to enable initiation of a subsequent activity that relies on the results of the completed workflow. As a further example, a customer can use the TIME scheduler to periodically probe the state of a remote service and install a callback that will update the state of a finite state machine representing the possible states of the remote services. By registering a callback on the transition from a nominal state to a state corresponding to a failure, the customer can automatically react to this event, without having to explicitly poll the current state of the remote service. This example demonstrates the flexibility offered by composition of primitives.

A `ComSubjectGroup` object is a specialization of `SubjectGroup` that allows a customer to define a boolean condition over events emanating from other `SubjectGroup` objects and receive notifications only when this condition has become true. The evaluation of the condition is incremental, which requires that the condition evaluator embedded in `ComSubjectGroup` objects distinguishes between three truth values: `TRUE, FALSE, UNKNOWN`. Since the condition only involves subject names, rather than state variables associated with events as in the case of the `SubjectGroupFilter`, the evaluator does not require the customer to specify any custom evaluator object. A `ComSubjectGroup` can, by virtue of composition, evaluate conditions that involve events from other `SubjectGroup` objects.

Finally, the *Aurora* infrastructure offers an `EventLoop` service that allows customers to subscribe to notifications for composite *hierarchical* events. Customers specify a composite event using the following *event path expression* notation:

$$ev_1 : p_1.ev_2 : p_2 : \ \ldots \ : ev_K : p_K,$$

where $ev_i, i = 1, \ldots, K$ are event name tags, and $p_i, i = 1, \ldots, K$ are optional predicates over the name/value pairs associated with the corresponding events. Assuming that the common case is for event notifications to be hierarchical in nature, with events encountered earlier in a path expression being of broader interest than following events which provide fine–grained detail, a customer can specify in detail when it should receive a notification. Path expressions (without predicates over name/value pairs) are also used by customers that announce events to this service. When an event is announced, only the customers that have subscribed with a path expression that includes the announcer's path expression as a prefix will be notified, provided that all their data–related predicates along the event path expression are satisfied. Subscribers to this service receive `Channel` objects, as in the case of the `SubjectGroup` family of services. The implementation of the service combines `SubjectGroupFilter` objects in a data structure that has the form of a collection of trees. Trees are formed when there are subscriptions with event path expressions that share a common prefix. Announcers of events to this service can be callbacks activated by any of the previously described variants of `SubjectGroup`, thus allowing for composition of all the event management services provided by the *Aurora* infrastructure. For example, an instance of `SubjectGroupFSM` may notify its clients when specific states are reached in the finite–state machine encapsulated in the `SubjectGroupFSM` object, where the state changes are signaled asynchronously by callbacks that have been registered with other event notification sources. Specific examples would be callbacks registered with time–related events (generated by the TIME scheduling service), callbacks registered with an instance of a `SubjectGroupFilter` service (that applies a filter to the data associated with event notifications), and callbacks registered for a path expression supported by an `EventLoop` service. Figure 4.5 illustrates this capability for composition.

The *Aurora* infrastructure also provides a service for persistent logging of event notifications in a relational database management system. This service allows a client to request that event notifications generated from an object of type `SubjectGroup` are monitored during a specified time interval, which may be open−ended. For each such event source, the persistent event monitor service maintains a table of all notifications received that includes the state data associated with the notifications. Clients can later retrieve series of records corresponding to notifications that occurred during a specified time interval, and have the option to filter such record series by applying a selection predicate over the name/value data associated with notifications. This service is part of the persistent logging and monitor service provided by the *Aurora* infrastructure, and aims to support applications that require a persistent audit trail for management purposes.

It is important to note that the event notification services of the *Aurora* infrastructure support multiple independent *event domains*. The goal is to avoid imposing a shared naming scheme on all potential workflow participants by allowing them to select the names of their published event notifications independently of all others. This functionality is required for being able to distinguish the event notifications generated within each domain, while maintaining the autonomy of the domain. An event domain is supported by a `SubjectGroup` factory service, and corresponds to a set of services offered by an independent service provider.

4.5 The HERMES Scripting Language

The *Aurora* run−time environment includes support for the HERMES language (described in Section 2.7) for configuring components into ensembles in order to construct the infrastructure for work sessions. We have developed an embedding of HERMES in the popular scripting language Tcl [Ous94], taking advantage of the extensibility features of the Tcl interpreter. We used Jacl, a Java−based implementation of Tcl, and added to this interpreter commands for enabling use of CORBA objects and commands for interacting with components and containers, including support for managing asynchronous requests. Furthermore, we added the commands listed in Table 4.5 for implementing event−driven flow of control, including support for ECA rules. This embedding of HERMES in Tcl

command	main parameters	return value
`rule_variable`	name, required flag, prompt handler	−−
`whenever`	trigger variables, action procedure	rule handle
`when`	trigger variables, condition evaluation procedure, action procedure	rule handle
`cancel_rule`	rule handle	−−

Table 4.5: HERMES/Tcl Commands for ECA rules and their Parameters.

allows a developer to combine procedural and event−driven control flow constructs, and supports component configuration as well as programming tasks that access distributed components and monitor/control asynchronous work requests.

The embedding allows direct interaction with CORBA objects using the OMG Name service to discover *interoperable object references (IORs)* and the OMG Interface Repository (IFR) to discover their interface for the purposes of invoking their exported operations via the *Dynamic Invocation Interface (DII)*. Interactions with these two basic infrastructure services can be time−consuming, therefore the embedding supports caching of IORs and results of IFR lookup operations as a performance optimization. Moreover, the embedding supports interaction with components as presented in Section 4.1, event management services as presented in Section 4.4, and component containers that support asynchronous requests over components as presented in Section 4.2.

Script variables can be set or updated by callbacks invoked as a consequence of event notifications. The commands `when` and `whenever` shown in Table 4.5 can be used in a script to declare that certain processing steps, grouped as a procedure, are to be invoked when a certain condition becomes true. Condition evaluation is triggered when a variable that has been declared to be a *rule–related variable*, using the `rule_variable` command, is set or updated by a callback or by direct assignment. The `rule_variable` command does not actually define the designated variable; rather, it informs the interpreter that upon executing commands that set or update the value of the designated variable there may be rules that need to be checked to determine if their associated actions are to be executed.

Using the `whenever` command, a script developer can specify that a procedure is to be activated upon a value change of a rule–related variable. Using the `when` command, a value change of a rule–related variable triggers the execution of a procedure that evaluates a condition and returns a value that can be either $+1$, 0, or -1, corresponding to truth values `TRUE, FALSE, UNKNOWN`. If the result of the condition evaluation procedure is `TRUE`, then the associated *action* procedure is executed. The TIME scheduling service can augment the rule constructs by triggering condition evaluation and/or actions based on time–related events, while the ECA scheduler and the other event–related services presented in Section 4.4 facilitate the use of complex evaluation conditions that involve correlation of events from distributed sources.

The flow of events in a distributed system consisting of loosely coupled autonomous components is inherently asynchronous, thus necessitating support for event–driven control flow constructs such as rules of the form `when` and `whenever` supported by the HERMES/Tcl combination that we have developed. However, there is a limitation in this model of execution with respect to the handling of missing values for variables required for the evaluation of activation conditions. Specifically, condition evaluation assumes that the variables referenced have been defined and their truth values are readily available for testing. This means that the distributed objects corresponding to the rule variables have been instantiated and activated. This approach is intuitive for applications where there is a flow of requests, each carrying its associated information elements, being routed among several services or processing components, but does not work as well for applications where information is not readily available and needs to be explicitly extracted from sources such as users, database systems or external services. In such cases, the rule processing infrastructure needs to be augmented with support for obtaining information from external sources, so that this information can then be used for condition evaluation.

In the *Aurora* infrastructure, a rule–related variable can be designated as *required*, so that, whenever its value is referenced in a condition a and its truth value is found to be `UNKNOWN`, a special action is triggered to obtain a value for it. Examples of such actions include prompting the user to enter a value, database queries, and computations that combine values obtained from several external services. The infrastructure assumes that components that can provide a value for a variable referenced in a rule export a uniform interface called `PromptInterface`, which offers the `get_value_for_variable()` method for requesting the (latest) value for a specified variable. This interface can be implemented by user interface components that directly interact with human workers, by wrapper services that encapsulate the details of interacting with a data source, or by components that combine data from several other sources to compute the requested value. Designating a rule–related variable as *required* requires that the script developer also specifies a component that exports this interface. When the evaluation of a condition needs the value of a *required* variable, the `get_value_for_variable()` method will be automatically invoked to obtain a value for this variable.

4.6 A Note on Performance Issues

To the best of our knowledge, there are no published data on the performance, availability, and scalability of commercially available workflow management systems. There are no standardized benchmarks such as the TPC suite for transaction processing applications [Gra91]. Concerns have been raised in the literature about the lack of adequate performance and scalability, and this can be expected to become a more pressing problem with the increasing deployment of workflow management systems in the context of Internet applications. There is still limited support for comprehensive workflow run–time environment monitoring, statistical data collection and reporting.

We believe that the design and adoption of meaningful performance benchmarks is a critical step towards building more scalable workflow management systems to sustain increasing demands in dynamic business environments, where interoperation is the norm rather than an exceptional case. Without widely accepted benchmarks, it is difficult to compare quantitatively a system's performance with that of another with a different architecture. We consider this a major stumbling block in the evolution of workflow management systems.

An important complication for workflow management benchmarks is that the time scale is different than that of other workloads [Den94]. In particular, the process of estimating performance metrics is complicated by the need to correlate the individual steps in units of work that involve both application programs and human participation in complex configurations. A consequence of this complexity is that it is hard to perform quantitative comparisons of the performance potential of alternative system architectures. So far, the predominant architecture is the client/server architecture outlined by the WfMC standards [Hol95]. However, the wide acceptance of Web–related technologies and distributed component frameworks provide opportunities for alternative designs. It is our view that by focusing on request management, rather than state transformations and transitions as in the WfMC reference model, it may be possible to obtain simplified but relevant performance models for evaluating alternative system architectures for workflow management in a distributed setting. Moreover, we advocate shifting the emphasis from a *production–centered* view of workflow, which implies low–level performance metrics such as average response times and throughput, toward a *satisfaction–centered* view, which focuses on commitments and conditions of satisfaction [MMWFF92]. This shift makes it possible to concentrate on metrics that are at the same level of abstraction as the business–level performance goals of workflow management. The service–level management approach presented in this dissertation provides an important link between the system and business aspects of workflow. It is our view that service–level management should be considered as the driving concern in the evaluation of system architectures for workflow management.

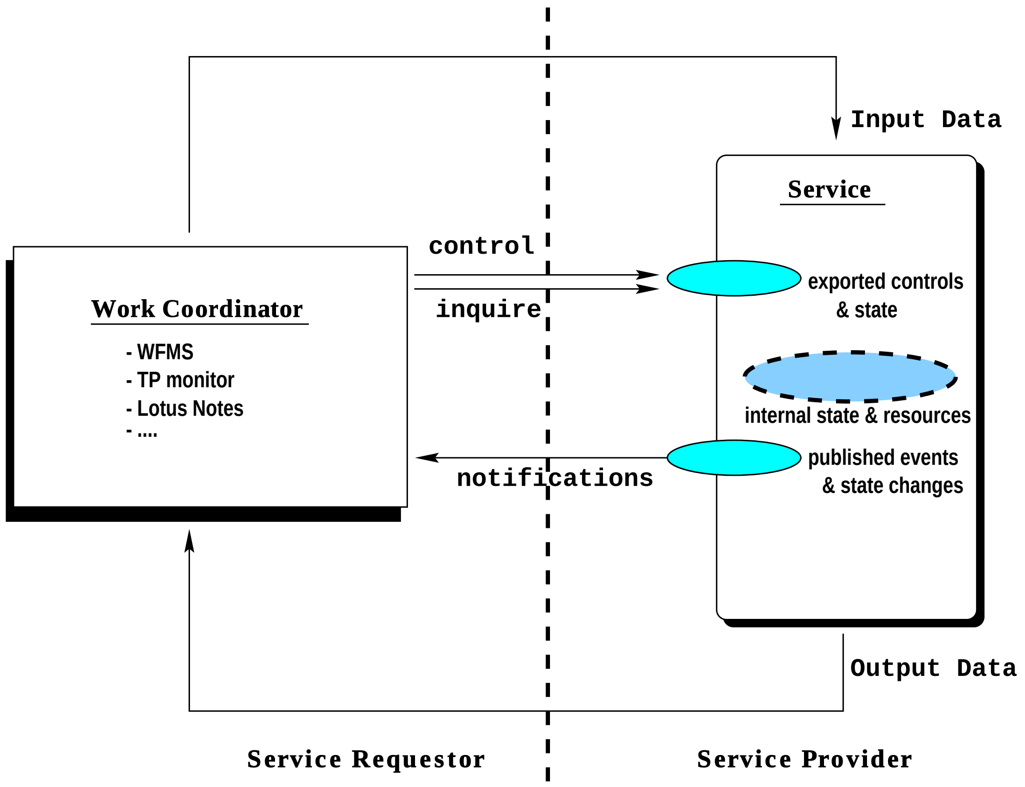


Figure 4.3: Interaction with a Component Hosted within a Container.

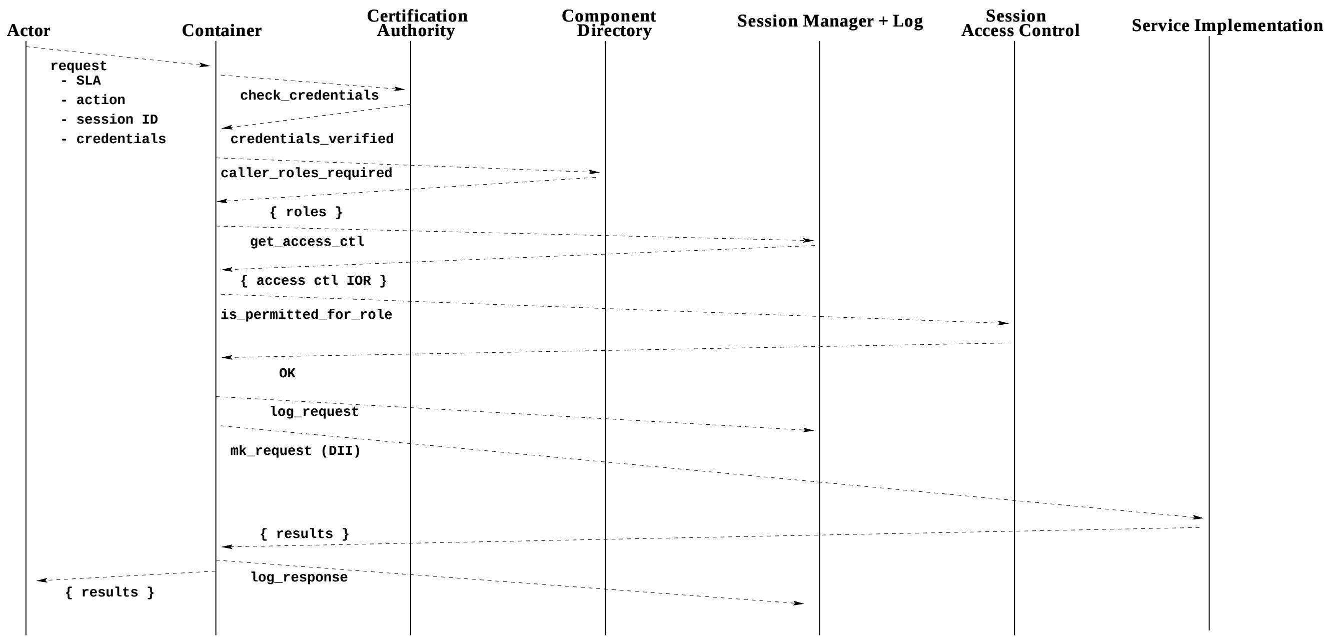


Figure 4.4: Access Control in the Context of a Work Session.

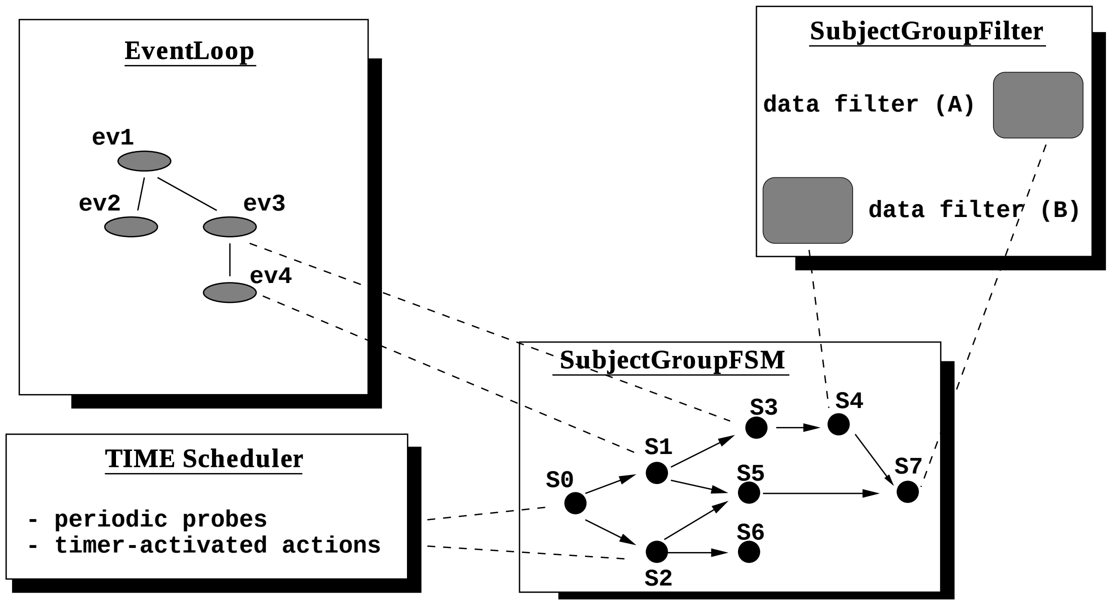


Figure 4.5: Example of composite event notifications by combining the `SubjectGroupFSM`, `TIME scheduler`, `SubjectGroupFilter`, and `EventLoop` services.

Chapter 5

Service level Agreements

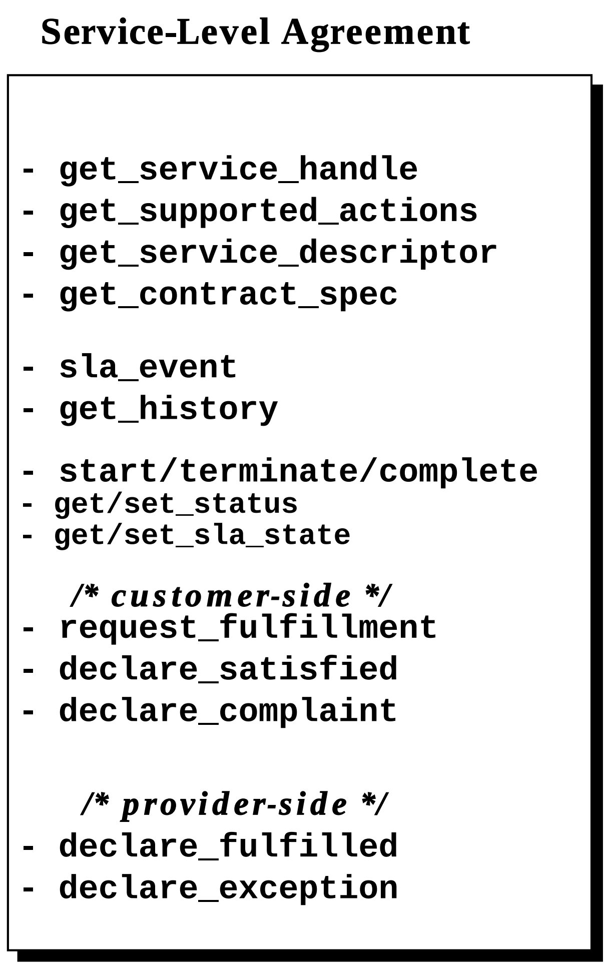
All process participants must voluntarily agree to co–operate with each other, perhaps on a restricted and temporary basis, and clearly define their commitments in a co–operative workflow. A service–level agreement documents the expected behavior of a service provider, in terms of functionality, performance and failure/exception handling behavior, for a particular client or class of clients. The material presented in this Chapter was first presented in [MPN98b].

5.1 Definition of Relationship Boundaries

The mechanics of controlling a service provider's interaction with a client is encapsulated in a first–class relationship object. Rather than attaching control information directly to the controlled service for interpretation in a service–specific manner, the service level agreement object enumerates the components that it controls, and encapsulates the state and code needed for monitoring and controlling interactions between the service provider and a customer. While it can be co–located with the service it controls, it is possible for the service level agreement to reside at a trusted third–party (such as a clearing house).

A service level agreement provides a reification of relationship boundary conditions for a service provider and its clients, together with a set of generic operations applicable to them. In effect, it represents a "contract" involving two or more parties and a set of "promises" that become effective once the contract has been accepted and all prerequisites have been fulfilled [RW97]. It provides the basis for monitoring and control of long–duration multi–step interactions, by authorizing actions, enforcing prerequisites and enforcing the execution of subsequent actions. It enables *ongoing proof of conformance* to agreed–upon service–level attributes, and enforces *accountability* as all actions covered by the service–level agreement can be persistently logged and later used in resolving disputes.

Figure 5.1 shows the basic service–level agreement interface provided by our framework. This interface, and its corresponding "factory" interface, are intended to serve as the inheritance base for specific process implementations, such as in the case of an electronic commerce scenario. It is important to note that the actual actions for implementing a service–level agreement are *generic* in the sense that the details of implementing an agreement are contained exclusively in the service provider's domain. The service–level agreement interface provides operations to inspect the current state in the performance of a service on behalf of a customer, and provides hooks for monitoring relevant event notifications. Introspection is supported by the `get_supported_actions` method,



Service-Level Agreement

```
- get_service_handle
- get_supported_actions
- get_service_descriptor
- get_contract_spec

- sla_event
- get_history

- start/terminate/complete
- get/set_status
- get/set_sla_state

   /* customer-side */
- request_fulfillment
- declare_satisfied
- declare_complaint


   /* provider-side */
- declare_fulfilled
- declare_exception
```

Figure 5.1: Elements of the service−level agreement interface.

which provides handles for `Action` objects that represent the basic capabilities provided by components implementing a service; namely, the right to use an exported interface, the right to register to receive notifications of an event, and the right to invoke a method. Moreover, the interface provides both human−readable and machine−readable descriptions of the terms and conditions governing the use of the service, as plain text and structured name/value pair list respectively. The `get_service_handle` operation provides a handle that allows the customer to invoke methods offered by the service through a container.

The actual sequencing of service performance is achieved by invocations of the methods `set_status`, `set_sla_state`, and `request_fulfillment`. The specific implementation defines the rules for determining when to permit updates of the state associated with this agreement. The method `request_fulfillment` allows a customer to issue a request for the service provider to perform the "promised" actions. This method returns a persistent request identifier computed by the container that will handle the request.

The method `declare_fulfilled` is to be called by the provider as a statement that all promised actions have been carried out successfully, whereas `declare_exception` covers the case when the provider declares that an exception occurred and needs to be handled according to the specific terms and conditions defined in the agreement. Similarly, the customer is expected to invoke `declare_satisfied` to explicitly acknowledge that a service request has been carried out with acceptable quality, or `declare_complaint` otherwise. Depending on the specifics of the agreement, `declare_exception` and `declare_complaint` may trigger compensating actions, programmed explicitly for each supported action. All events of relevance to an agreement are persistently logged, and can be reviewed for auditing purposes. Events of particular significance include the instantiation of the agreement, its termination (either successfully or unsuccessfully), the state/status changes during its life–cycle, and the declarations made by the independent participants.

5.2 Logging and Monitor Infrastructure

The term *workflow* denotes [Moh97] all operational aspects of a business process, including the sequence of tasks and who performs them, the information flow to support tasks, and tracking and reporting mechanisms that measure and control tasks. Workflow management aims not necessarily to automate all tasks of a process, but rather to automate the tracking of states of tasks and to allow specification of preconditions to decide when tasks are ready to be executed and of information flow between tasks. Current state–of–the–art workflow systems [AAAM97, GHS95] are mainly concerned with the routing and assignment of tasks, providing little support for administration and management tasks, such as workflow monitoring and reporting, management of resources, tracking the status of ongoing processes, and exception handling. Support for such tasks is essential for establishing a robust and manageable work environment, and providing quality guarantees. Overall business planning and operations control require comprehensive mechanisms for performance monitoring and policy enforcement. Such mechanisms improve *accountability*, which entails that the availability and level of performance of all entities involved in workflow processing be tracked and maintained according to predetermined levels.

The need for management support is exacerbated in dynamic open environments, where services provided and managed by multiple autonomous authorities need to be integrated. Such environments significantly stretch the assumptions underlying current workflow system designs, especially in the areas of autonomy and dynamic control flow. The World Wide Web (WWW) [T. 94] is rapidly becoming a universal information and communication resource, thus creating a pressing need for workflow management technology to address the requirements of conducting business processes on the WWW. It is necessary to extend workflow execution models, and their underlying infrastructures, beyond the current state–of–the–art to support more dynamic and adaptive processes, and provide more effective support for human–intensive work. As argued in [She97], this can be achieved by integrating coordination and collaboration technologies with information management. We consider this integration as the basis for realizing a shared workspace, enabling collaboration among participants in a work session by allowing participants to invoke services and publish results.

This dissertation proposes *service level management* as a framework for addressing all managerial issues in an integrated manner, in the context of a workflow execution model that supports dynamic configuration of work sessions in a dynamic run–time environment. The information contained in the SLA is required for providing guidance to clients of a service as to what is the service's expected behavior, and can be retrieved from the *Aurora* repository. It can be used during the binding phase to select among alternative service offers, based on the client's requirements on attributes of the service. Dynamic open environments such as the Internet are inherently unreliable and exhibit widely varying responsiveness. However, provided that a service implementation supports cancellation of the effects of service actions, it is possible to modify or cancel the effects produced by a work session, by executing *compensating actions*. Clients are allowed to use each service only through the SLA exported by the service provider, which enforces the access policies specified by the provider and implements the functionality and behavior "promised" by the service interface. As containers are instantiated at run–time, a run–time representation of the SLA to be enforced for a client is instantiated at run–time, to monitor and control action requests.

Information about expected performance and supported compensation actions guides clients in planning a strategy for obtaining service despite failures and unpredictable performance. For example, a client can use the information exported by the SLA to set timeouts and to schedule retries and compensating actions in case of failure. Moreover, a client can abort/cancel its requests when the measured service–level parameters (such as transfer rate and response time) become worse than specified thresholds. These capabilities contribute to making services in a dynamic open environment more predictable, and, in this sense, more manageable. Furthermore, service providers can modify SLAs at run–time, by updating their service offer entries in the repository, to reflect their updated access policies and current performance level.

By informing clients about how to obtain information on the supported guarantees for transactional execution and expected performance, this specification contributes to making services in a dynamic open environment more predictable, and, in this sense, more manageable. It is important that the autonomy of service providers is not compromised, as a service provider is the only authority responsible for exporting an interface for use by clients and for establishing and enforcing service attributes such as transactional and performance guarantees. Another important point is that a service provider may combine multiple services, made available by other autonomous providers, in order to provide a *composite service* to a client. A client does not need to be aware of this complexity. Moreover, other service providers may not be aware that their services are being used in the context of a composite service request. The SLA exported by a service provider hides such implementation aspects, exporting only aspects related to the service level that a client can expect, together with information about available "emergency" actions to compensate for actions that were not completed successfully or that the client wishes to revoke. This paradigm, therefore, takes into account the characteristics of dynamic open environments.

Service level management has long been practiced in enterprise data processing centers [Noo89], typically on mainframe computing systems, and has recently begun to draw attention in the context of client/server business systems. However, there is yet no such support for open environments such as the Internet and the WWW. Apart from the difficulties of monitoring management metrics from multiple heterogeneous systems, in open environments there is no single authority responsible for all available services. In particular, there is no single authority responsible for maintaining integrity as in the case of traditional transaction processing systems. Multiple authorities (service providers) need to interact to implement all the steps in a session. Each service provider is responsible for its own services and may have no knowledge of the interactions among authorities that take place in a session. As argued in [DP97b], different participants may have differing views of the boundaries of a session, as it is possible for a provider to hide from its clients whether services from other providers are utilized in the course of a session.

Service level management requires a comprehensive monitoring infrastructure. *Aurora* provides an infrastructure for building such environments, and, through its monitoring mechanisms that allows each service provider to log information about its own state and its interactions with others, supports monitoring of pair−wise interactions between parties. A session may span multiple distributed resources, owned by autonomous providers. Keeping track of the activities of tasks is achieved by requiring each container to register with the logging system that is part of its run−time environment. Thus, the logging systems of session managers constitute the basis of a distributed monitoring infrastructure. The *Aurora* monitor enables a client (for example a workflow administrator using a management application) to collect all log records about events of interest to the execution of a workflow. This infrastructure enables tracking the progress and current state of service flows, as well as maintaining the interaction history for each participant. A basic function of this infrastructure is support for *correlation* of event records. Thus, it is possible to examine the entire *interaction history* for a session, the records related to the actions of a particular participant, or the records related to a particular service provider. Another important function is to support aggregation queries, especially based on temporal information (such the completion time of tasks). This is important for benchmarking the performance of the workflow infrastructure, and producing concise performance reports.

Log records can simply define the start and end of steps in a work session, or provide more detailed information that can later be used for compensating for certain actions. Such information is expressed as a list of attribute−value pairs, and may include, for each task (processing step) in a session, the name of the *resource* used, the start and ending time, the persistent identifier for the work session within which the task is executed, and various performance metrics of relevance for the task and session (such task completion time). A basic function of the monitor service is support for *correlation* of log records. Methods are provided to examine the entire *interaction history* for a work session, retrieve the records related to a particular task within a session, and select the records that satisfy a client−specified predicate over the attribute−value pairs associated with log records. Queries to the monitor service can be further qualified by specifying a time interval, as log records include a timestamp. Figure 5.2 illustrates the structure of the *Aurora* monitor. As multiple groups within IT are responsible for different parts of the IT infrastructure (such as applications, databases, networks, desktop and server machines), it is difficult to maintain a single−system image for management purposes, resulting in *multiple, overlapping views*, each focusing on specific infrastructure components without fully considering inter−component dependencies. A user request may span multiple management views, as it may involve multiple components. The *query and correlation engine* of the *Aurora* monitor (currently under development) enables grouping of log records, gathered from multiple session managers, according to several criteria. Examples include grouping of all records related to a given session participant, grouping of all records related to a given work session, and grouping of all records related to a given service provider.

Monitoring requires applications and resource managers to provide notification of events, as well as mechanisms for accessing application state variables. Service offers published through the repository service allows service providers to specify the metrics that can be requested from an application component, and the events (with their associated attributes) that a component can generate. The entry for a resource in the repository includes all the essential information that enable monitoring and control of the component.

Management applications, acting as clients of the monitor service, may invoke the `GetRecs, GetRecsByConstraint, GetAllRecs` methods in order to correlate log records, produced by multiple tasks in the context of a work session and stored in multiple logging systems. This is essential for keeping audit trails of critical business processes, collecting performance−related data to identify bottlenecks, as well as for enabling flexible recovery and compensation in the event of failures that cause exceptions. Recovery and compensation are possible since the producers of log records can provide sufficient state information to enable a management application to cancel or modify the effects of an action, by including in their log records the name and arguments of each action method that they invoke. This information can later be used by compensating actions invoked through the SLA.

The specification of the measurements and events that a component can generate complements the specification of the component's functional interface. This allows service providers to selectively expose implementation details and run−time state to clients, thus enabling monitoring performance metrics and control of operational parameters. The actual implementation of monitoring and control is by instrumentation provided by component developers. The SLA hides such details, providing a uniform interface for clients. The uniform container provided by the *Aurora* architecture encapsulates, apart from application components (which incorporate instrumentation for monitoring and control), a run−time representation of the supported SLA, which includes, the name of the service, the specification of the service interface (in the form of attributes and action methods), the access control restrictions for each actions, information about how to cancel, or compensate for, the effects of each action (if this is possible), and information about expected performance (such as the expected average and standard deviation of response times). Management applications can discover at run−time (via the operations `ListStateVars, ListControlOps` supported by the uniform management interface of Table 4.1.8) what state variables are exposed and what control operations are provided by each of the components, including state variables that record performance−related information. Performance−related information provides a more predictable view of services to clients, as it can provide guidance in setting time−outs at the client−side, and assist in planning an "access strategy". The latter is important for fully automated sessions involving coordinated access to multiple services.

The SLA documents the *expected behavior* of service providers, for a given client or client class. SLA enforcement requires on−line monitoring of the delivered service levels and the actual resource/service demands, and the ability to invoke configuration and control actions to affect the behavior of active tasks. *Ongoing proof of conformance* to a SLA requires the ability to produce on−line reports on the delivered service levels, thus achieving accountability. This aspect is particularly important for business processes that span organization boundaries.

5.3 Integration with the Component/Container Framework

An important aspect is that the service–level agreement framework readily integrates with the component/container framework, so as to allow interactions between customers and services from autonomous providers to take the form of contract fulfillment processes. Such contracts are specified in the cases where it is important to be explicit about the mutual obligations of interacting transactions and the handling of violations of agreements on access terms and level of service. In such cases, the service–level agreement object encapsulates a semantic agreement between the customer and the service provider, sequencing their interaction and maintaining a persistent log of the actions invoked and their outcomes. Figure 5.3 shows how a service–level agreement object mediates a customer's interaction with a component that implements a service offered by a provider. The service–level agreement object encapsulates a reference to the component implementing the service, which is not directly available to the customer. The customer interacts with the service only through the service–level agreement object, by invoking the `request_fulfillment` method. In turn, the service–level agreement object interacts with the container that hosts the component, using the uniform request management interface exported by the container. All steps in the interaction are persistently logged by the service–level agreement object, which can also receive notifications from the service provider's component about the progress of a request. An interaction is considered to have been completed successfully only when the provider has invoked the `declare_fulfilled` method *and* the customer responds by invoking the `declare_satisfied` method.

The service–level agreement implementation, as instantiated by the corresponding run–time object, dictates the specific semantics of interaction between a service provider and a customer. Specific examples are given in the case study presented in Chapter 7. Provided that a domain model of the specific application area is available, it is possible to develop comprehensive service–level agreement objects. The generic framework presented in this dissertation handles low–level details of reliable state tracking, and of automatic response to deviations from the expected run–time behavior. It is also important to note that a service–level agreement object may be hosted either directly by a service provider *or* by a third–party authority that acts as auditor of the interactions between providers and their customers.

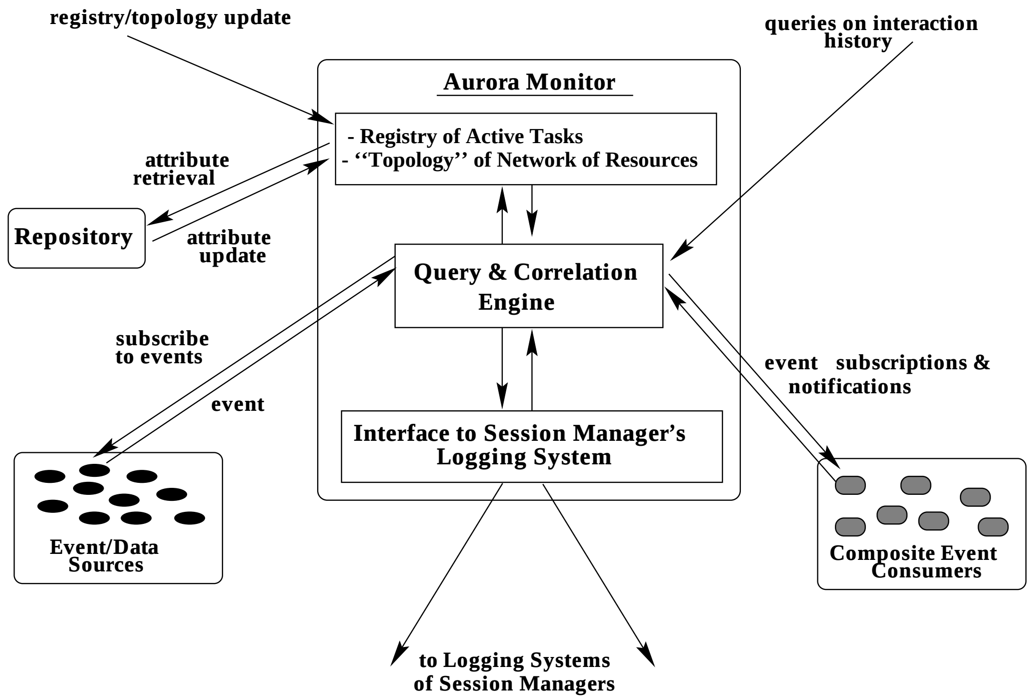

registry/topology update
queries on interaction history
Aurora Monitor
- Registry of Active Tasks
- "Topology" of Network of Resources
attribute retrieval
Repository
attribute update
Query & Correlation Engine
subscribe to events
event
event subscriptions & notifications
Interface to Session Manager's Logging System
Event/Data Sources
Composite Event Consumers
to Logging Systems of Session Managers

Figure 5.2: Structure of the *Aurora* Monitor.

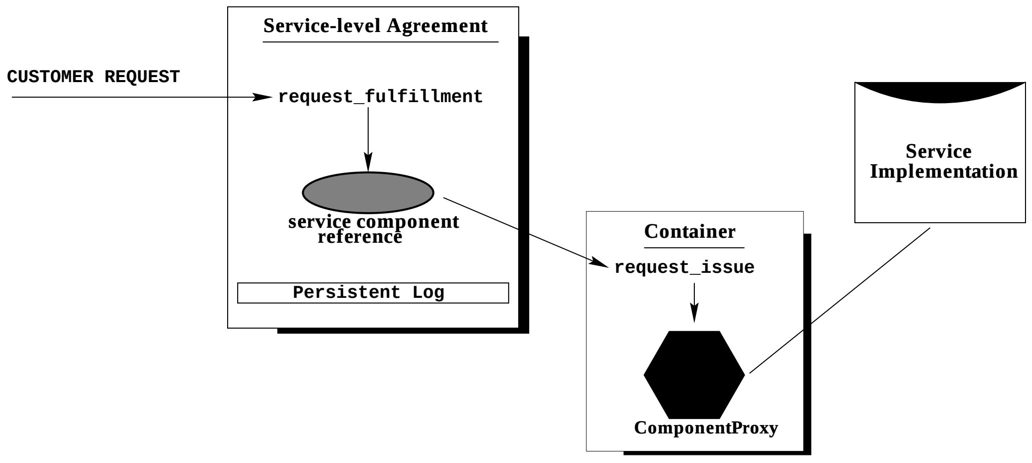


Figure 5.3: Service−level agreements as entry points to services.

Chapter 6

Work Session Framework

Service–level agreements are explicitly instantiated in the run–time environment as objects. Such objects control the sequencing of interactions between the customer and the provider, maintain persistent state for the current status of requests, and allow both the provider and the customer to declare explicitly what they see (independent of each other) as satisfactory outcomes or as deviations from the agreed–upon execution pattern. Service–level agreements can be used directly, or in the context of *work sessions*, which are collections of distributed resources that can be accessed under the restrictions of an access policy that is enforced by an access controller object specified by the authority owning the collection.

The *Aurora* component platform provides a framework that adopts the *satisfaction–oriented* view of workflow presented in [MMWFF92] (see Figure 6.1), where commitments, conditions of satisfaction and timely completion are the guiding concerns. This is different from the focus of current workflow management systems on managing state transformations and transitions, and flow of *information products* (such as business documents) between workflow participants, which are usually human workers that perform tasks by using assistant tools. It is our view that a satisfaction–oriented perspective on workflow is more appropriate for generic *services*, made available by autonomous providers, as it does not require full exposure of internal state transformations and transitions, and also does not assume that the results from steps in a workflow are discrete units, such as information products. Moreover, this view of workflow is more appropriate for open environments as no globally consistent state needs to be maintained.

Instead of focusing on the flow of data objects and their life–cycle within a closed system, the thesis presented in this dissertation is to adopt a satisfaction–oriented approach, focusing on managing and tracking requests by customers to service providers. Requests are essentially *tokens* by which customers demand that providers perform the steps necessary for realizing their service–level commitments, as documented in service–level agreements. The combination of service–level agreements with a generic resource sharing environment is a major contribution of the research effort presented in this dissertation. The work session framework was first presented in [MPN98a].

Figure 6.3 illustrates the basic objects defined in the *Aurora* work session framework. This framework models workflows that may span administrative domains as configurations of participants in shared workspaces that group together the resources used in the workflow. Shared workspaces, also termed *work sessions*, define the boundaries of a process by encapsulating the resources, participants, tasks with dependencies, work requests and related events that occur during its life–cycle. The *Aurora* work session framework allows participants to browse through the resources in a shared workspace, select the resources that they consider of value, and interact with them, through their exported proxy objects that are hosted in containers and can be used through the asynchronous request management interface described in Section 4.2.

Figure 6.2 outlines the overall model of process control in the *Aurora* run−time environment. Three types of actions, corresponding to roles, are clearly distinguished and may be fulfilled at run−time by independent authorities. A *process definition* comprises specifications of the set of resources, essentially components that export service interfaces, that need to be instantiated and appropriately interconnected in order to fulfill a certain goal (such as offering a composite service to customers). A *process instance* exists in a (distributed) run−time environment that hosts the components specified by a process definition, and realizes inter−component communication paths and sequencing constraints. The *Aurora* infrastructure described in this dissertation provides such a distributed run−time environment. The task of configuring and interconnecting components, as well as the task of controlling the sequence of service invocations and handling asynchronous event notifications, are supported by the HERMES scripting language that is part of the *Aurora* infrastructure. The HERMES scripting language can also be used for monitoring and control of process instances, by accessing and manipulating the state variables exposed by components, and by requesting notifications of asynchronous events related to significant state transitions. Alternatively, application developers can directly use the unform management interfaces exported by *Aurora* containers to monitor and control the components that they host.

6.1 Resources, Tasks and Work Sessions

A *session resource* is a wrapper for the container−generated reference to a component that implements one or more services. This component can be accessed through the container's uniform management interfaces (described in Chapter 4). A session resource has descriptive properties and unique security credentials. It can potentially be used in performing a task, in the context of a session, but it is important to note that it exists independently of any task or session and operates autonomously under the authority of its owner. An access control policy, represented by an access controller object of type `ACL` (see Section 4.3), specifies the access restrictions for the resource. This policy is interpreted by the container upon attempts to invoke actions exported by the resource's underlying component. A resource can belong to one or more sessions, which take the form of resource collections maintained by independent participants. A *work session*, which is a resource in itself, can contain sub−collections of resources, in a hierarchical configuration, and may enforce access restrictions of its own.

A specialization of the *session resource* interface represents a *session participant*, which can create sessions, initiate tasks, and accept work requests. A *task* is a specialization of the *session resource* type that has a designated owner and dependencies on resources, which can be either *predecessors* or *successors*. In other words, a task relies on a set of resources (which can be tasks in their own right), and is used by another set of resources (which may be the results of executing the task). Predecessor resources are expected to provide required input for further processing, while successor resources receive the results of such processing.

Tasks may be short−lived, to support a specific short−term request by a participant, or open−ended, providing a service that combines a number of (predecessor) resources. Open−ended tasks exist independently of their owners. A participant can create a task, by interconnecting resources, and then other participants can use this assembly of resources for their own purposes, subject to the access restrictions enforced by the task object *and* the work session that contains the task.

Tasks, sessions, and participants are capable of receiving resource−related event notifications. Such notifications allow the participants of a work session to keep track of changes in the availability of resources in the shared workspace of a session. The addition and removal of resources in a work session triggers notification via callback invocations. The *Aurora* session manager service allows potential session participants to browse through the currently available resources in a work session, and also make their own resources available for sharing by registering them in the session. A work session is essentially a dynamic collection of shared resources, which are managed exclusively by their owners. The *Aurora* work session framework only requires that the owners of resources add their resources to the collection of the session. Access to resources (specifically, the right to use an interface exported by the component whose reference is encapsulated in a `SessionResource`) is restricted by the access controller associated with the session. Session participants can create sub−collections of resources, with their custom access controllers, and register them with an existing session, under their own terms as encapsulated by their access controllers.

Figure 6.4 shows a schematic of a work session that can be modeled using this framework. This example shows that a work session may involve both humans and software systems, and requests issued to a certain service provider may be decomposed into subrequests, which in turn can be processed either by the provider or delegated to other providers, in a manner transparent to the customer that issued the original request. Different participants in a workflow may thus have differing views of the boundaries of interactions, without compromising the overall integrity of the process or the integrity of individual participants. This allows a recursive, scalable construction of complex workflows, *without compromising the autonomy of service providers*.

6.2 Differences from Other Approaches

Initiatives such as SWAP [Swe98] and OMG jointFlow [jFl98, Sch99] mainly address the submission of tasks to distributed enactment services, but do not address issues such as the hierarchical decomposition and coordination required for workflow among autonomous workflow engines, and service−level management. Figure 6.5 illustrates the structure of the reference model defined by the Workflow Management Coalition (*WfMC*), which comprises the vast majority of workflow system vendors. This model corresponds closely to the predominant architecture of current−generation workflow systems. Examples of widely−deployed commercial systems conforming to this reference model include FlowMark [LR94, Flo96] and InConcert [MS93]. As shown in Figure 6.5, the reference model relies on a *monolithic* server that is responsible for the major workflow management functions, including process enactment, management of the staff directory, binding of activities to participants, distribution of work items to the work−lists of workflow participants, work−list management, and invocation of application tools for use by participants. An implicit assumption in this reference model is that the workflow execution engine is the principal and authoritative co−ordinator of processes, which are either completely contained within its sphere of control or execute under the control of another workflow engine that supports interoperation by implementing an appropriate interoperability interface. Therefore, an enterprise's organizational model needs to be represented in detail using the system's modeling functionality and all applications involved in the workflow need to be adapted to allow invocation and control by the workflow engine. Changes and evolution of processes can therefore entail major development efforts, especially if they involve the integration of resources from external organizations. By relying on a monolithic server that is responsible for both workflow coordination and activity execution, current workflow management systems as defined by the WfMC impose severe limitations on flexibility and scalability. [PPC97a] criticizes the consequences of the current systems' rigid structure. A major problem is that work lists cannot be shared by heterogeneous workflow engines, since they are not externally accessible. Thus, in order for a participant to take part in multiple workflows on different workflow servers, a separate work list needs to be maintained at each server and client applications need to maintain multiple dedicated connections. Moreover, three different interfaces are used for assigning work to human participants, invoked applications, and sub–workflows on other servers. This lack of transparency in how the activities are implemented makes delegation of work difficult, as the workflow application is inherently dependent on the choice of the actual implementation of each activity.

Such shortcomings are exacerbated in open systems such as the Internet where interconnected and interdependent components are expected to be able to handle interactions that do not adhere to predefined scheduling constraints. Autonomy considerations imply that assumptions and convenient arrangements about communication channels, exported functionality, access policies, performance, and exception handling behavior cannot be taken for granted, but rather have to explicitly established, in a case–by–case fashion for each resource of interest. With severely limited access to the internal state and operational procedures of independent service providers, workflow applications need to become flexible enough so as to accommodate *service–level agreements* that define the mutually accepted terms and conditions for interaction, within well–defined boundaries of authority that respect the authority of independent participants.

In our approach, we shift the emphasis on defining a generic asynchronous request management infrastructure, and complement that with a flexible work session framework and service–level agreements as first–class objects. Under the severe restrictions imposed by the autonomy of resource owners, the *Aurora* infrastructure can by necessity only handle aspects of distributed workflow related to request management and data interchange (in the form of request parameters and responses). Support is provided for asynchronous event notification, and persistent logging of steps in the processing of requests, allowing the participants of work sessions to monitor their interactions and perform audit checks, independently of each other. By having service–level agreements explicitly represented in the run–time environment, the relationship between service providers and their customers can be explicitly monitored and managed on–line. This approach is a departure from current practices, which do not represent the terms of interaction so explicitly. Moreover, by providing several workflow support services for general use (such as the access control framework and asynchronous event notification services), rather than centralizing such services in a workflow server, our approach contributes to a more open infrastructure for large–scale distributed workflow applications, without requiring multiple accounts and work lists to be maintained. Finally, by focusing on asynchronous request management, rather than the low–level details of interoperability interfaces for specific types of work performers, our approach allows a generic treatment of service providers and their interactions with customers that does not comprise the autonomy of workflow participants. What is more, this approach to workflow allows the integration of service−level agreements in workflow processing by simply encapsulating references to service−level agreement objects within session resource objects.

The focus on asynchronous request management is also characteristic of the *Simple Workflow Access Protocol* (SWAP), described in an IETF Internet Draft [Swe98]. SWAP allows a client to initiate, control and monitor asynchronous long−duration service execution at remote sites. *SWAP* provided the basis for the *Interoperability Wf−XML Binding* that was issued by the Workflow Management Coalition in January 2000 [WfX00]. This specification defines an XML−based language for modeling the data transfer requirements of interoperable workflow systems. Related work is also reported in [PPC97b] that presents an architecture for supporting workflows involving autonomous participants and service providers, based on asynchronous requests among workflow participants and standard interfaces for *sources* and *performers* of work items. The work presented in this dissertation extends this previous work by introducing a framework for service−level agreements, that represent explicit commitments by providers to customers on service−levels.

The OMG jointFlow specification [jFl98, Sch99] adapts the WfMC runtime standards to a business objects execution environment. The jointFlow specification distinguishes between requesters, that implement the `WfRequester` interface to allow a process to propagate status updates, and work performers, that implement the `WfActivity` interface to allow observation and control of the state of an activity in the context of a workflow. Objects that implement the `WfProcess` interface represent entities that perform work requests, often by delegating them to other entities, providing operations to control the execution of the work request and to observe its state. The specification also includes the `WfEventAudit` interface for describing the contents of status−change events produced by `WfActivity` and `WfProcess` objects.

A workflow−process model can be translated into the jointFlow metamodel in two ways: One option is to use the framework as an object veneer on top of an existing workflow engine; in this case the implementation of the jointFlow interfaces would delegate most of their operations to the back−end WFMS. Alternatively, the process definition could also be coded into a `WfProcess` object and and a set of `WfActivity` objects that are hosted by a business object server.

The jointFlow specification is a major evolutionary step for WfMC−compliant workflow management systems. However, the specification does not address the hierarchical scaling and coordination required for a flexible, distributed co−operation among autonomous workflow engines. It is assumed that all the parties involved in the overall workflow are integrated through the jointFlow metamodel, with `WfActivity` objects acting either as *adapters* for existing business objects or as *bridges* for interaction between a main workflow and another workflow application. In the latter case, the `WfActivity` implementation is expected to also implement the `WfRequester` interface, so as to allow the workflow activity to receive status updates from the sub−workflow. There is no support for cases where the workflow systems may not be able to communicate status information to one another. More importantly, there is no support for monitoring service levels, and in particular monitoring of exceptions and deviations from the expected behavior of service providers. Another subtle limitation on the autonomy of providers is that the life−time of `WfActivity` objects is tightly coupled with that of `WfProcess` objects. In effect, a `WfActivity` object cannot exist outside the context of a `WfProcess` object. In contrast, in the *Aurora* work session framework `SessionResource` objects exist independently of work sessions, under the control of their respective service providers. `SessionResource` objects can thus be discovered at run–time by potential customers, whereas in the OMG jointFlow framework it is assumed that in order to initiate a `WfProcess` all the required resources have been resolved (in a manner that is not covered by the specification, which only provides the `WfAssignment` interface representing the association of a resource with an activity) and appropriate `WfActivity` objects can be instantiated. By placing no restrictions on the life–cycle of session resources, the *Aurora* work session framework respects the autonomy of service providers and provides mechanisms for managing asynchronous requests and reliable tracking of interaction state through service–level agreements. Requests are not explicitly represented in the OMG jointFlow framework, and therefore cannot be monitored and managed in detail. Moreover, service–level management issues are not addressed, although they become crucial in the case of inter–organizational workflow which presumes the existence of *contracts* among the different participating organizations. A main theme in this thesis is that such contracts should be explicitly represented at run–time in order to improve accountability and monitor conformance to the terms of co–operation agreed upon by the participating organizations.

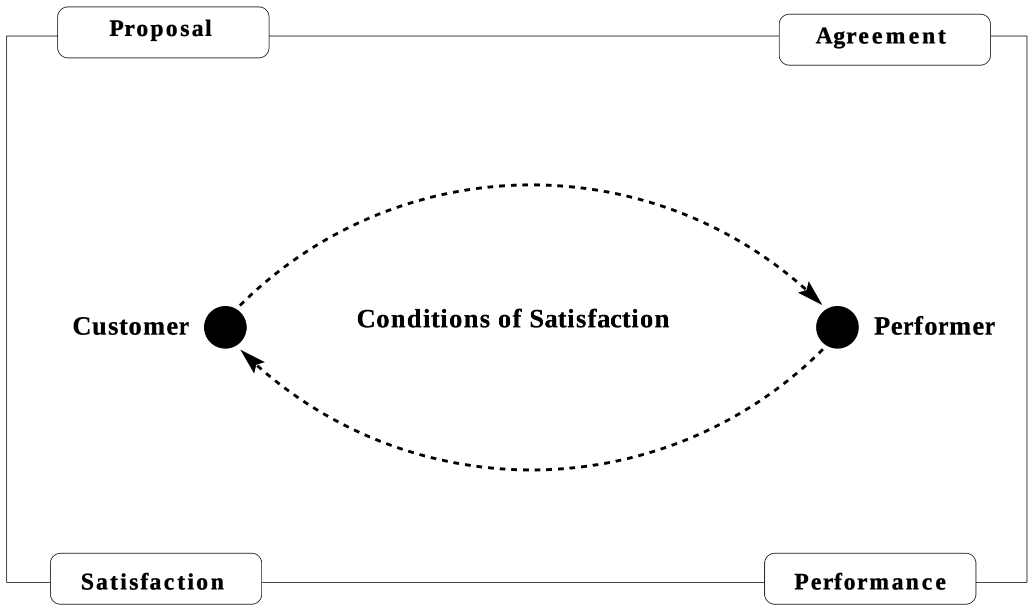


Figure 6.1: A Satisfaction−Oriented View of Workflow.

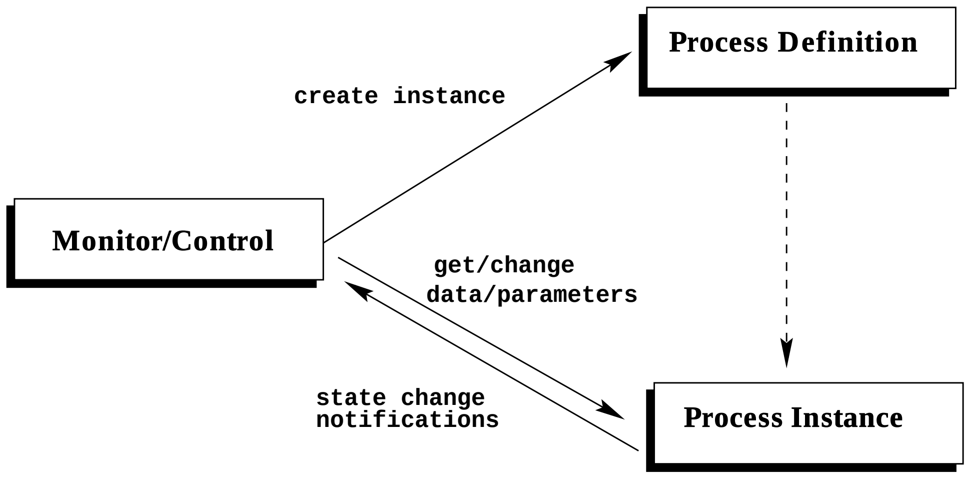


Figure 6.2: Instantiation and Control of Process Instances in the *Aurora* run−time environment.

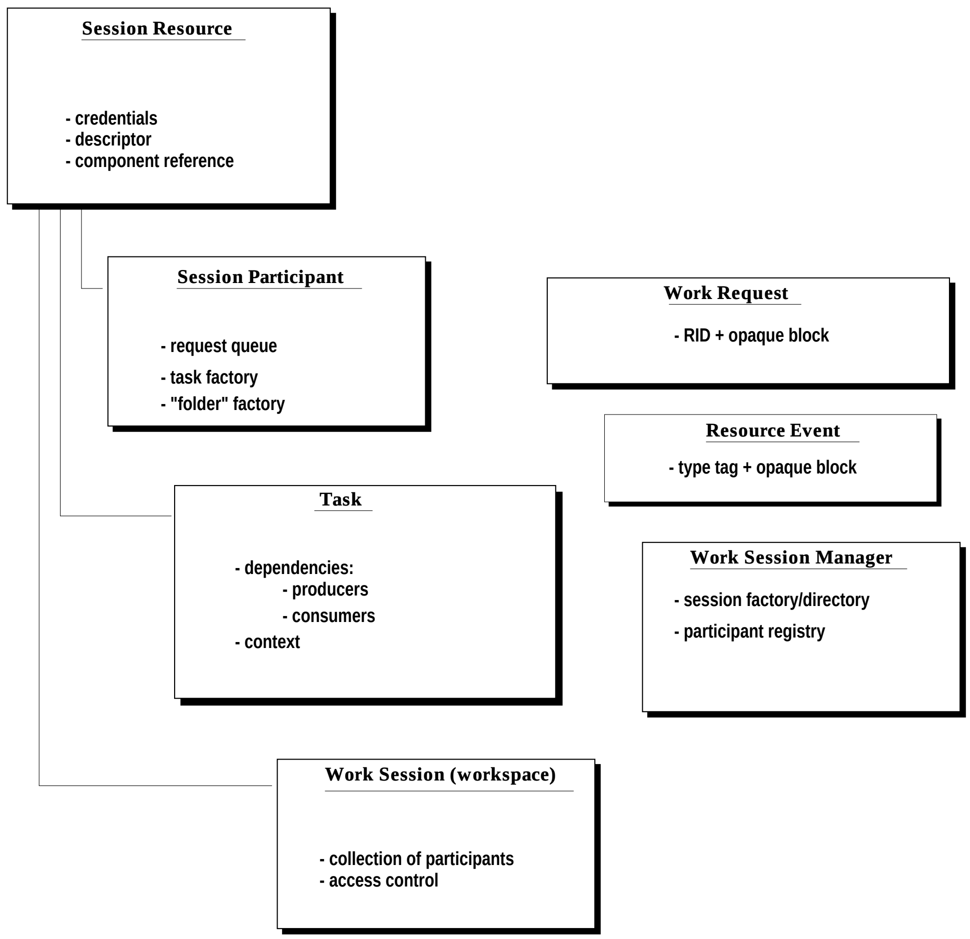


Figure 6.3: Elements of the *Aurora* session framework.

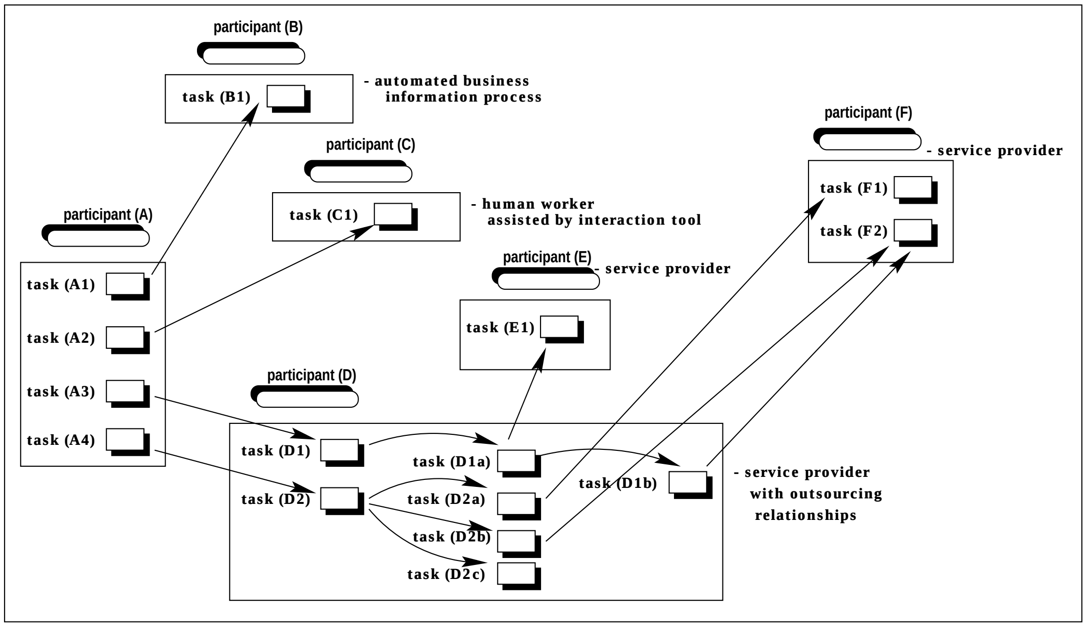


Figure 6.4: Example of a work session involving autonomous participants.

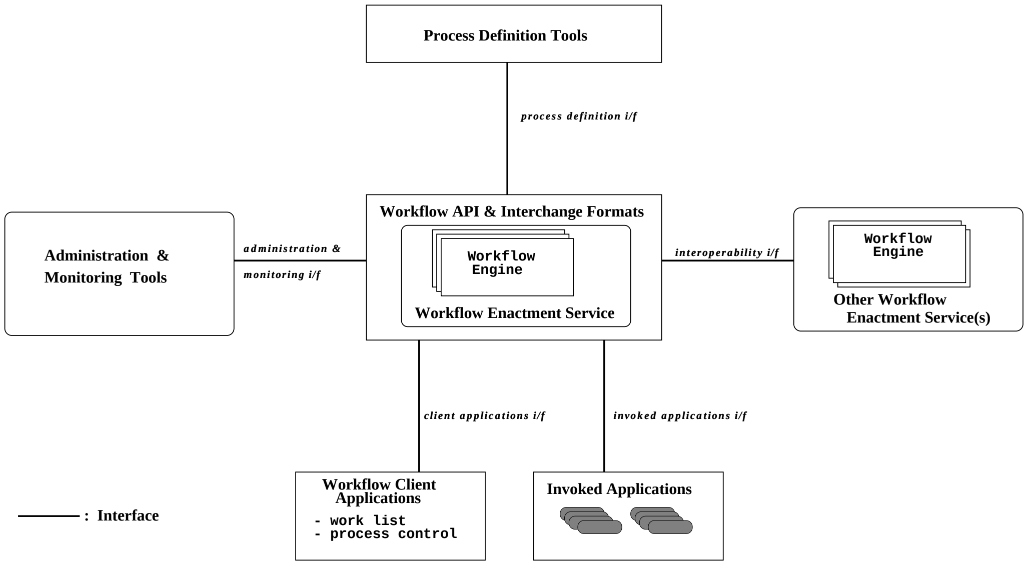


Figure 6.5: The Workflow Management Coalition Reference Model.

Chapter 7

Case Study

This chapter describes a prototype implementation of an electronic commerce system that was developed for the purposes of demonstrating the integration of service−level agreements in the run−time environment. The prototype also provides the basis for a qualitative comparison of the work session framework presented in Chapter 6 to the approach taken by current−generation workflow management systems, as reflected by the jointFlow specification by the OMG Business Object Domain Task Force [jFl98].

The electronic commerce system consists of components, as described in Chapter 4, that provide a number of commerce−related services. The components are combined to build a distributed electronic commerce environment, assuming that these services are offered by independent authorities and therefore mandate as loose a coupling as possible. These components were used in the two alternative implementations, the one based on the *Aurora* infrastructure and the one based on the OMG jointFlow framework. The two alternative implementations highlight the limitations of the OMG jointFlow framework, as discussed in Section 6.2.

7.1 Description of Components

The suite of components developed for this case study offer the following services:

- *ProductsCatalogue*: a database of product descriptions
- *Inventory*: extension of *ProductsCatalogue* that keeps track of the available quantity for each product and generates *alerts* when the inventory level for a specified product becomes lower/higher than a threshold
- *ShoppingBasket*: service for combining products from one or more *ProductsCatalogue* services into a single order
- *PaymentProcessor*: front−end for electronic payment processor
- *OrderTracking*: order management and tracking service capable of handling orders that include multiple products from independent providers
- *FeedbackLog*: service for logging client feedback and problem reports so that providers can inspect them and take appropriate actions to handle them

The implementations of these components rely on a number of support services provided by the *Aurora* infrastructure, such as the persistent data storage service, the volatile data storage service, and the logging service.

7.2 Order Processing Workflow

We focus on order processing and tracking in an electronic commerce environment, built using the components described in Section 7.1. This workflow is described by the state transition diagram of Figure 7.1, which defines the life−cycle of an order. The `VOID` state

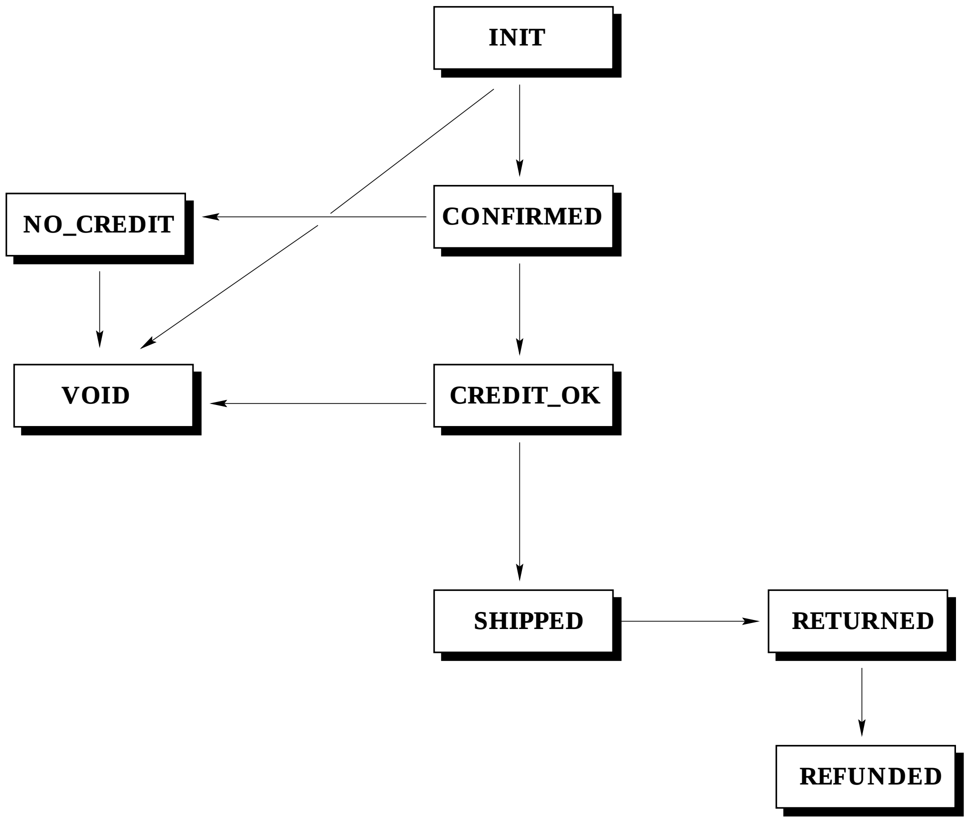


Figure 7.1: Order life−cycle in the e−commerce case study

represents an order that has been cancelled for some reason, either by the customer or by a provider. After an order has been submitted (`INIT` state), the order management and tracking service performs validity checks on the product and shipping information submitted by the customer to determine if the order can be marked as `CONFIRMED`. This step is followed by checks of the payment−related information submitted by the customer to determine if order−processing can proceed (`CREDIT_OK` state) or should stop (`NO_CREDIT` state). The order management and tracking service marks an order as `SHIPPED` after all the products selected by the customer have been shipped to the specified address, provided that this action takes place within a specified deadline. Until this deadline expires, the order may be modified or cancelled either by the customer or by the providers, with no consequence. After the products have been shipped, the customer is given the option to return the products (`RETURNED` state), in which case the order has to be marked as `REFUNDED` within a specified deadline.

The order processing and tracking workflow described by Figure 7.1 is inherently event−driven. The prototype uses the `SubjectGroupFSM` service and the `EventLoop` service, described in Section 4.4, to register callbacks to be invoked when an order reaches each of the states shown in Figure 7.1.

7.3 Integration using the *Aurora* Framework

We developed `ComponentProxy` objects (see Section 4.1) for the components presented in Section 7.1, so as to allow them to be combined while remaining independent, under the control of autonomous service providers. A work session was established, comprising of `SessionResource` objects that provided persistent handles for the `ComponentProxy` objects, which in turn were hosted by containers. The order processing workflow was implemented by executing a configuration script, expressed using the scripting language presented in Section 4.5, that established callback actions to be executed upon notifications on order state changes.

The resulting component configuration had minimal coupling among components, since the component model, in particular the standard interfaces `Importer`, `Exporter`, and `Notifier` facilitated composition for the purposes of combining the functions supported by the components, without implicit embedding of object references in the component implementation code. For example, the order tracking service can retrieve a reference for the `PaymentProcessor` service by querying its `Importer` interface, allowing flexibility in the selection of the service. Moreover, by providing implementations of the `ComponentControl` interface for each of the components, it is possible for system administrators to monitor the components at run–time, by inspecting the state variables that they expose. In the prototype implementation, components exposed measurements of the average and maximum response times for requests. Additional measurements could easily be accommodated in the framework, provided that the component implementation exposed these measurements to the corresponding `ComponentControl` objects.

An important point is that the implementation based on the *Aurora* work session framework benefits from the service–level agreement framework. This framework allows the providers of services such as the order tracking system and the payment processor to explicitly declare their terms of service, and track the course of fulfillment transactions jointly with their customers. This functionality covers an aspect of workflow management that is outside the scope of the OMG jointFlow specification. Section 7.5 describes examples of interaction sequencing through a service–level agreement object.

A further characteristic of the implementation based on the *Aurora* work session framework is the loose coupling and dynamic binding of resources/tasks, which is facilitated by the uniform management interfaces exported by components in the *Aurora* run–time environment, and the directory service that allows service providers to publish the interfaces and their associated non–functional properties (access control restrictions, compensation support, expected performance). Service providers may register or withdraw their resources at any time, while potential customers that express their work requests in terms of abstract specifications using the resource type definitions in the service directory need not be aware of such decisions by the service providers. Customers and service providers remain autonomous, provided that a facility such as the service–level agreement framework explicitly delimits their interactions.

The implementation based on the *Aurora* work session framework represents tasks and work requests as first–order entities, existing outside the scope of specific work sessions. Task are modeled as entities that can consume and produce resources, so as to explicitly model dependencies among services. Work requests are submitted to session participants, which are also explicitly represented in the context of work sessions. Session participants, being resources themeselves, can process work requests by issuing further requests to other participants and by utilizing resources available in the current work session context.

7.4 Integration using the OMG jointFlow Framework

For the purposes of experimentation with alternative designs for the work session framework, we developed an integration framework based on the system based on the OMG jointFlow specification [jFl98], and used it as an integration platform for the components presented in Section 7.1 to implement the order processing workflow scenario. This specification [Sch99] adapts the WfMC run−time standards to a business objects execution environment, defining a framework for implementing distributed workflow applications. It defines standard interfaces for enabling interoperation of process components, monitoring of process execution, and associating workflow components with resources.

In short, the OMG jointFlow framework mainly addresses the submission of tasks to distributed workflow enactment services, but does not address issues such as the hierarchical decomposition and coordination required for workflow among autonomous workflow engines, and service−level management. As discussed in detail in Section 6.2, there is no support for cases where the workflow systems may not be able to communicate status information to one another; it assumed that objects of type `WfActivity` represent tasks in the overall workflow process

7.5 A Demonstration of Service−level Agreements

In Sections 7.3 and 7.4, two implementations of the same workflow management application were described, using the *Aurora* work session framework and a framework based on the OMG jointFlow specification, respectively. This section describes how a service−level agreement has been integrated in the prototype electronic commerce system. Specifically, the prototype supports monitoring of the delay in shipping an order once it has been confirmed, and automatic invocation of compensating actions in case of deadline expiration. This functionality relies on the TIME scheduling service, described in Section 4.1.6.

The service−level agreement object implemented for this demonstration uses the `SubjectGroupFSM` service to keep track of state transitions in the processing of an order. Completing an order involves submitting a shopping basket, containing products from one or more product catalogues, to an order tracking service. The service−level agreement object encapsulates a reference to a container−hosted component that implements the order management and tracking service. The customer does not directly access the order management and tracking service, but rather interacts with the service via the service−level agreement object that supervises order processing. The actual steps in processing an order are carried out by a service of type `OrderTracking` that is hosted in the provider's run−time environment. The service−level agreement object interacts with a container to relay asynchronous requests for order processing. The parameters of these asynchronous requests include the deadline for completing order processing, and the compensating actions, encapsulated in an object that exports the `compensation` method, to be *automatically* triggered in the event of missing the deadline. In the prototype, the compensating actions are merely to notify both the customer and the provider. More elaborate actions could easily be accommodated, provided that they are coded as part of the `compensation` method.

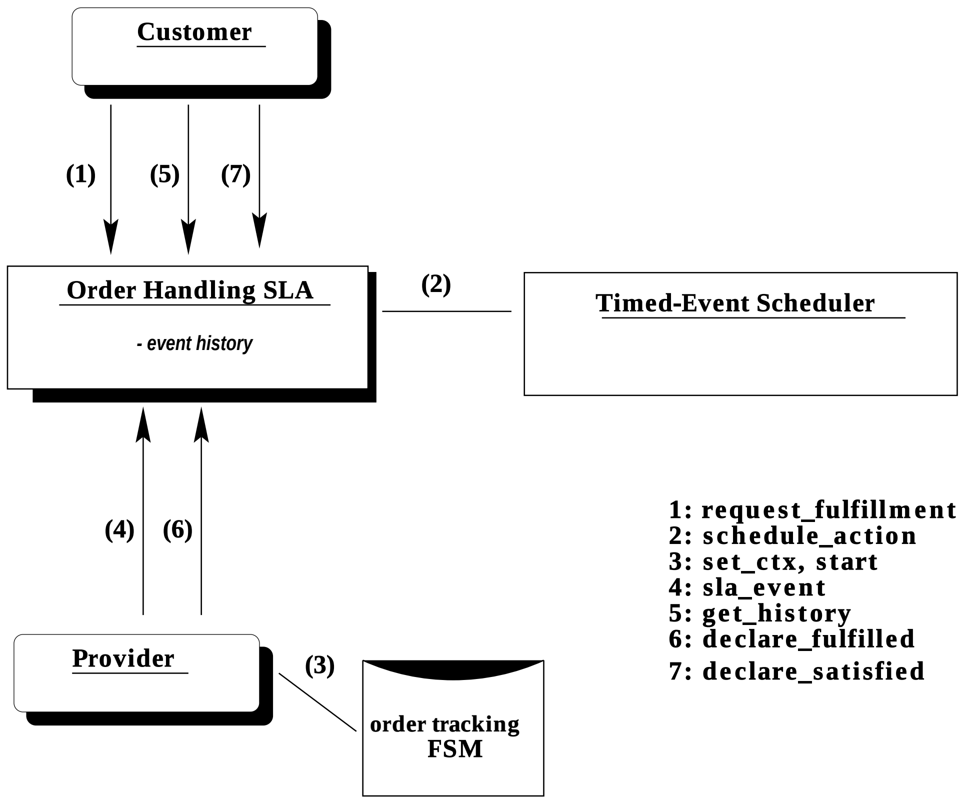


Figure 7.2: Service–level agreement for shipping an order: Normal case.

Figure 7.2 illustrates the flow of requests and events in the case of an order that is shipped within the specified deadline.

1. Customer invokes `request_fulfillment` method on the service–level agreement object, which invokes the `submit_order` method of the `ShoppingBasket` service, through an asynchronous request. The parameters of the `request_fulfillment` method include the security credentials of the customer. The return value of this method is the persistent identifier for the request for executing the `submit_order` method. All parameters and return values of methods invoked on the service–level agreement object are persistently logged, and are associated with the security credentials of the invoking entity.

2. The service–level agreement object invokes the `schedule_action` method on the TIME scheduling service, to arrange for receiving a notification when the deadline for shipping the order expires.

3. The methods `set_ctx` and `start` are invoked by the service provider, to mark that the execution of the service has begun, and to initialize the persistent state maintained by the service–level agreement object.

4. While processing the order, the service provider can emit asynchronous event notifications, using the `sla_event` method. These notifications, mostly progress indications, are persistently logged.

5. The customer can view the event history for an ongoing request, by invoking the `get_history` method on the service–level agreement object. It is important to note that the service–level agreement object is not necessarily hosted by the service provider, which may keep his own log of events and actions. The log maintained by the service–level agreement object is the externally visible record of the ongoing interaction, and serves as proof of conformance or deviations from the behavior promised by the provider.

6. The service provider performs, at his site, the steps necessary for handling the order. Assuming that the deadline for shipping the order is not exceeded, the service provider at some point in time invokes the `declare_fulfilled` method. This invocation is an explicit declaration that the service provider claims to have completed his part of the agreement.

7. The interaction is completed successfully when the customer, independently of the provider, confirms that the service was indeed completed in an acceptable manner. This is achieved by invoking the `declare_satisfied` method on service–level agreement object.

Figure 7.3 illustrates the flow of requests and events in case the deadline for shipping a confirmed order expires. The first five steps are the same as in the case of normal execution flow.

1. Customer invokes `request_fulfillment` method on the service–level agreement object.

2. The service–level agreement object invokes the `schedule_action` on the TIME scheduling service, to arrange for notification in case the deadline is missed.

3. The methods `set_ctx` and `start` are invoked by the service provider.

4. While processing the order, the service provider can emit asycnhronous event notifications, using the `sla_event` method.

5. The customer can inspect the event history by invoking the `get_history` method.

6. Assuming that the deadline expires without the provider having completed order processing, the TIME scheduling service notifies the service–level agreement object. This is achieved by an invocation of the callback method `timed_event_cb`. This method persistently logs the fact that the deadline has expired.

7. After logging the deviation from the promised execution behavior, the service–level agreement object invokes the `compensation` callback method on the service provider.

8. Optionally, the service provider may acknowledge the deviation, by emitting an event to be logged by the service–level agreement object. It should be noted that the service–level agreement object does not directly affect the internal state of the service provider's order processing system.

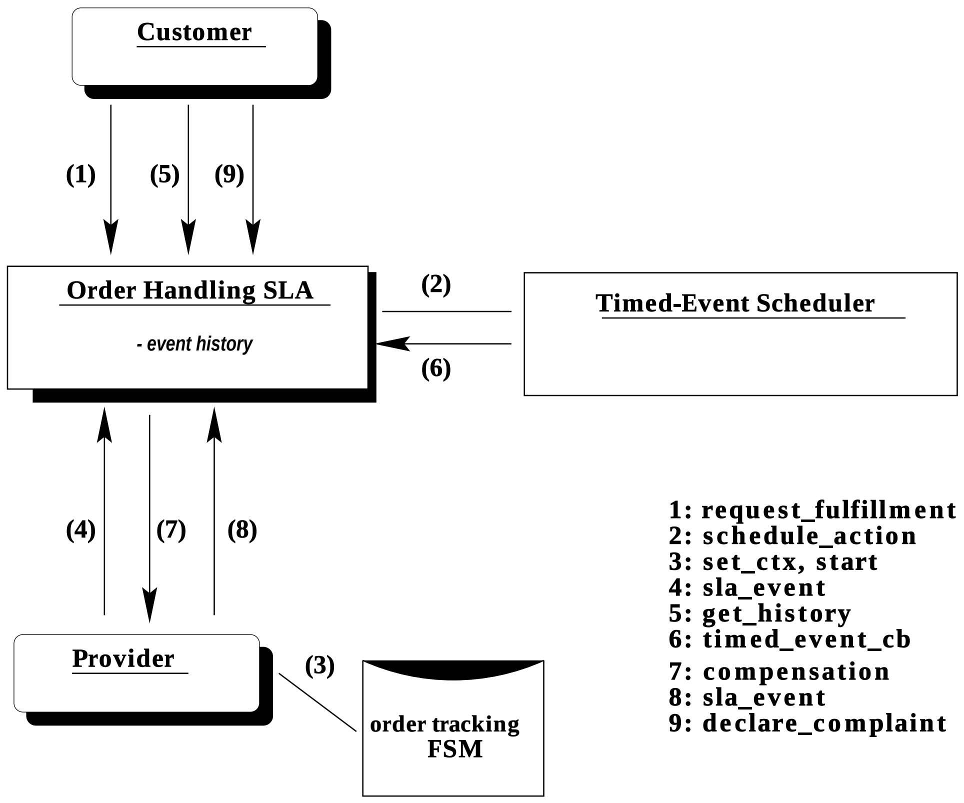


Figure 7.3: Service–level agreement for shipping an order: Exception handling.

9. The customer may opt to log a formal complaint, using the `declare_complaint` method. This action would be useful for auditing purposes and handling disputes. Depending on the specific terms of the service–level agreement, the provider may continue processing to ship the order, at a reduced cost, or the order may be cancelled.

It should be noted that, in both examples, the service–level agreement object keeps track of the current state in the prcoessing of the order. This is achieved by having registered a callback to be activated when the provider's `SubjectGroupFSM` service emits notifications. Since notifications are generated when each state is reached, as well as when specific state transitions take place, the implementor of the service–level agreement can program specific reactions for specific events of relevance to the order processing workflow.

Chapter 8

Conclusions and Perspective

8.1 Summary of Results

This dissertation presented a model for network–centric applications that rely on the cooperation of autonomous services in an open distributed environment. Extensions to the CORBA object framework were presented in Chapter 4 to facilitate composition of new services by combining existing ones, with support for on–line monitoring and control. Service–level agreements were introduced into the distributed component infrastructure in Chapter 5 to allow interactions while preserving autonomy and guaranteeing accountability. By constraining both the service provider and the service customer to interact through service–level agreements, which are first–class objects in our infrastructure, the run–time properties of services become more predictable and in this sense more manageable. An infrastructure that utilizes components and service–level agreements to model configurations of resources and tasks in the context of work sessions provides a framework for dynamic and adaptive workflow, presented in Chapter 6. In our perspective, work sessions are defined primarily by the (dynamic) collection of resources made available by autonomous service providers, the requests issued by the partners involved, and the mutual commitments between service providers and their customers implied by these requests through explicit agreements on service–level aspects. Thus, the *Aurora* infrastructure not only provides support for implementing the actions taken by individual service providers to satisfy their commitments, but also explicitly supports co–ordination and tracking of requests for performing actions. A case study from the domain of electronic commerce, presented in Chapter 7, demonstrates the integration of service–level agreements in the run–time environment. The case study also provides the basis for a qualitative comparison of the work session framework presented in Chapter 6 to the approach taken by current–generation workflow management systems, as reflected by the jointFlow specification by the OMG Business Object Domain Task Force [jFl98].

The work session abstraction, that models a Web of resources utilized by interdependent tasks, contributes to realizing a *shared workspace* within which diverse applications and tools can be shared by a community of end–users in dynamic configurations. This workspace enables interoperation of components that implement services, while respecting the autonomy of the service providers, in a sense enabling dynamic trading in the context of a virtual *marketplace* [Der97]. In such a setting, composition capabilities are essential for service providers to gain market advantage, by allowing more rapid creation of new services. Moreover, such a setting requires a comprehensive infrastructure for allowing independent providers to express terms and conditions for usage of their services, including access restrictions and performance−related indicators and controls, and support for both providers and customers to monitor the conformance of ongoing interactions to mutually agreed terms. The research presented in this dissertation is a step towards realizing such a vision.

8.2 Directions for Further Research

All in all, we consider system−level infrastructure support for service−level agreements to be a major direction for research, as well as an essential requirement for advancing the scope and quality of processes in a dynamic open environment, especially in the context of *electronic commerce* applications. We are particularly interested in applying our work in *enterprise portals* that combine infrastructure, business models and organizational structures to enable network−centric business processes. Such systems provide the means for connecting customers, partners, suppliers and employees, by integrating the business processes of individual participants and building business communities. Portals encompass applications to support functions such as discovery of information and business opportunities, collaboration, business transaction execution, customer support, and supply chain management. Many of these applications may share software components, and require an infrastructure that readily supports on−demand combination of available service offerings to create customized packages for customers. This infrastructure needs to support dynamic deployment of new components and services, dynamic configuration, and service−level monitoring, despite the challenges raised by the large−scale distribution and autonomy of process participants. It is our view that these requirements can be addressed through the development of a *service−level monitor* that, in analogy with a *transaction processing monitor* [GR93], provides a controlled run−time environment and support services for dynamic workflow involving distributed, independently developed and managed components. The *Aurora* infrastructure is a step towards this direction.

An important development is the increasing popularity of *thin−client devices*, such as personal digital assistants (PDAs) and third−generation mobile phones with Internet access capabilities. Use of such devices is becoming pervasive, and there is growing interest in using them as generic service access terminals. Their inherent limitations, namely limited CPU and memory capacity, limited bandwidth, and intermittent bandwidth, pose hard challenges for application development. It is our view that the service−level agreement and work session frameworks presented in this dissertation could be used in this context as well. However, the infrastructure services presented in this dissertation rely on CORBA and Java technologies that are too heavy−weight for thin−client devices. A direction for further research is the design and development of light−weight mechanisms for allowing thin−client devices to access services from independent and autonomous providers. It is our view that the Wireless Application Protocol [Ltd99] by the WAP Forum, an international industry consortium, offers a major enabling technology for allowing generic service access from thin−client devices. A direction for research is to design and develop a *service access gateway* for services hosted by the *Aurora* run−time infrastructure that will allow WAP−compliant thin−client devices to access services.

Another major direction for research is to investigate how to handle changes in the service–level agreements exported by service providers, especially in the case of interdependent providers. For example, assume that service provider A relies on the services of providers B and C, and that each of the providers exports its own service–level agreement, $sla(A)$, $sla(B)$, and $sla(C)$, respectively. If service provider B changes its service–level agreement, for example by promising shorter response times or by offering more elaborate compensating actions, service provider A needs to be notified, since its service–level agreement must take into account the changes made by service provider B. In the case of changes related to expected response times, it may be possible to automatically update the parameters of the service–level agreement exported by service provider A. In the case of compensating actions and access restrictions, changes may be much more difficult to handle. Due to the increasing interdependencies between service providers, the problem of propagating changes in service–level agreements needs to be investigated in–depth.

Moreover, we are interested in applying the service–level agreement framework in cases where an ontology models the specific application domain. In this setting, service–level agreements can encapsulate application domain–specific semantics, allowing for more detailed handling of the interaction between service providers and their customers. By giving an explicit link between the systems–level and business–level aspects of the application infrastructure, service–level agreements provide a flexible framework for explicitly representing and managing complex relationships between service providers and their customers, and allow for reliable tracking of the interaction party by all sides involved, without comprising autonomy. It is for this reason that we believe service–level agreements should become an integral part of large–scale distributed application platforms.

Bibliography

[AAA+95] G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan, M. Kamath, and R. Guenthoer. "Exotica/FMQM: A Persistent Message–Based Architecture for Distributed Workflow Management". In *Proc. IFIP Working Conference on Information Systems Development for Decentralized Organizations*, pages 1––18, 1995. Also available as IBM Research Report RJ9912, IBM Almaden Research Center, 1994.

[AAAM97] G. Alonso, D. Agrawal, A.El Abbadi, and C. Mohan. "Functionality and Limitations of Current Workflow Management Systems". *IEEE Expert*, 1(9), 1997. Special Issue on Cooperative Information Systems.

[All97] C. Allen. "WIDL: Automating the Web with XML". *World Wide Web Journal*, 2(4):229––248, 1997.

[BCGP97] M. Baldonado, C.K. Chang, L. Gravano, and A. Paepcke. "The Stanford Digital Library Metadata Architecture". *Int'l Journal of Digital Libraries*, 1(2), 1997.

[BDS+93] Y. Breitbart, A. Deacon, H.J. Schek, A. Sheth, and G. Weikum. "Merging application–centric and data–centric approaches to support transaction–oriented multi–system workflows". *ACM SIGMOD Record*, 22(3), 1993.

[BE95] E. Berkel and P. Enrico. Effective Use of MVS Workload Manager Controls. IBM Corp., 1995.

[Ber96] P.A. Bernstein. "Middleware: A Model for Distributed System Services". *Communications of the ACM*, 39(2):86––98, February 1996.

[BMR96] D. Barbara, S. Mehrotra, and M. Rusinkiewicz. "INCAs: Managing Dynamic Workflows in Distributed Environments". *Journal of Database Management*, 7(1):5––15, 1996. Special Issue on Multidatabases.

[Bos97] J. Bosak. "XML, Java, and the Future of the Web". *World Wide Web Journal*, 2(4):219––227, 1997.

[BSCHL97] I. Ben–Shaul, A. Cohen, O. Holder, and B. Lavva. "HADAS: A Network–Centric Framework for Interoperability Programming". *International Journal of Cooperative Information Systems*, 6(3/4):293––314, 1997.

[BTJW98] D. Beringer, C. Tornabene, P. Jain, and G. Wiederhold. "A Language and System for Composing Autonomous, Heterogeneous and Distributed Megamodules". In *Proc. Int'l Workshop on Component–Based Modeling of Distributed Systems (in conjunction with DEXA'98)*, pages 826––833, 1998.

[CC96] IBM Corp. and Hewllet−Packard Corp. “Application Response Measurement API User Guide”, 1996. Available via URL http://www.hp.com/go/ARM.

[CCD$^+$96] K.M. Chandy, A. Chelian, B. Dimitrov, H. Le, J. Mandelson, M. Richardson, A. Rifkin, P.A.G. Sivilotti, W. Tanaka, and L. Weisman. “A World−Wide Distributed System Using Java and the Internet”. In *Proc. 5th IEEE Int'l Symposium on High Performance Distributed Computing*, 1996.

[CD97] L. Cardelli and R. Davies. “Service Combinators for Web Computing”. In *Proc. USENIX Conference on Domain−Specific Languages*, 1997.

[CGS97] S. Ceri, P. Grefen, and G. Sanchez. “WIDE: A Distributed Architecture for Workflow Management”. In *Proc. 7th Int'l Workshop on Research Issues in Data Engineering*, 1997.

[COR94] *“The Common Object Request Broker: Architecture and Specification”*. Object Management Group, Framingham, Mass., 1994. Revision 2.0.

[Den94] P. Denning. “The Fifteenth Level”. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1−−4, 1994.

[Der97] M.L. Dertouzos. *“What Will Be : How the New World of Information Will Change Our Lives”*. Harper San Francisco, 1997.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. “Organizing Long−Running Activities with Triggers and Transactions”. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 204−−214, 1990.

[DP97a] A. Dan and F. Parr. “An Object Implementation of Network Centric Business Service Applications (NCBSAs): Conversational Service Transactions, Service Monitor and an Application Style”. In *Proc. OOPSLA'97 Business Object Workshop*, 1997. Available via URL http://www.tiac.net/users/jsuth/oopsla97/.

[DP97b] A. Dan and F. Parr. “The Coyote Approach for Network−Centric Service Applications: Conversational Service Transactions, a Monitor and an Application Style”. In *Proc. High Performance Transaction Processing (HTPS) Workshop*, 1997.

[Elm92] A. K. Elmagarmid, editor. *Transaction Models for Advanced Database Applications*. Morgan Kaufmann, 1992.

[FFH$^+$98] S. Field, C. Facciorusso, Y. Hoffner, A. Schade, and M. Stolze. “Design Criteria for a Virtual Market Place (ViMP)”. In *Proc. European Conference on Research and Advanced Technology for Digital Libraries*, pages 819−−832, 1998.

[Flo96] “IBM FlowMark Programming Guide (Version 2 Release 3)”, 1996. Document Number SH19−8240−02.

[FNGD92] D. Ferguson, C. Nikolaou, L. Georgiadis, and K. Davies. "Goal Oriented, Adaptive Transaction Routing for High Performance Transaction Processing Systems", 1992. IBM Research Report RC18139.

[FNGD93] D. Ferguson, C. Nikolaou, L. Georgiadis, and K. Davies. "Satisfying Response Time Goals in Transaction Processing Systems". In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, 1993.

[Fuc96] M. Fuchs. "Let's Talk: Extending the Web to Support Collaboration". In *Proc. 5th Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 1996.

[GEK+94] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavarupu. "Falcon: On−line Monitoring and Steering of Large−Scale Parallel Programs". Technical Report GIT−CC−94−21, Georgia Institute of Technology, 1994. Available at http://www.cc.gatech.edu/.

[GHS95] D. Georgakopoulos, M. Hornik, and A. Sheth. "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure". *Distributed and Parallel Databases*, 3(2):119−−154, April 1995.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Java Series. Addison−Wesley, September 1996.

[Gol90] C.F. Goldfarb. *"The SGML Handbook"*. Oxford University Press, 1990.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Gra91] J. Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann, 1991.

[GSBC99] D. Georgakopoulos, H. Schuster, D. Baker, and A. Cichocki. "Managing Process and Service Fusion in Virtual Enterprises". *Information Systems, Special Issue on Information Systems Support for Electronic Commerce*, 24(6):429−−456, 1999.

[GSM+95] K. Goldman, B. Swaminathan, T.P. McCartney, M.D. Anderson, and R. Sethuraman. "The Programmer's Playground: I/O Abstractions for User−Configurable Distributed Applications". *IEEE Trans. on Software Engineering*, 21(9):735−−746, 1995.

[Gun95a] N.J. Gunther. "Flight Recorders, Timing Chains, and Directions for SPEC System Benchmarks". *SPEC Newsletter*, March 1995.

[Gun95b] N.J. Gunther. "Thinking Inside the Box and the Next Step in TPC Benchmarking − A Personal View". *TPC Quarterly Report*, January 1995.

[Hol95] D. Hollingsworth. "The Workflow Reference Model", 1995. WFMC−TC−1003 − available via WWW at URL http://www.wfmc.org.

[jFl98] "OMG BODTF RFP #2 Submission Workflow Management Facility", 1998. OMG Document Number bom/98−06−07.

[KGMP97] S.P. Ketchpel, H. Garcia–Molina, and A. Paepcke. "Shopping Models: A Flexible Architecture for Information Commerce". In *Proc. Int'l Conference on Theory and Practice of Digital Libraries*, pages 65––74, 1997.

[Kic96] G. Kiczales. "Beyond the Black Box: Open Implementation". *IEEE Software*, 13(1), 1996.

[KS95] N. Krishnakumar and A. Sheth. "Managing Heterogeneous Multi–System Tasks to Support Enterprise–Wide Operations". *Distributed and Parallel Databases*, 3(2):155––186, April 1995.

[KTL$^+$92] H. Krasner, J. Terrel, A. Linehan, P. Arnold, and W.H. Ett. "Lessons Learned from a Software Process Modeling System". *CACM*, 35(9):91––100, 1992.

[KZLO98] E. Kaldoudi, M. Zikos, E. Leisch, and S.C Orphanoudakis. "PLEGMA: An Agent–Based Architecture for Developing Network–Centric Information Processing Services". Technical Report FORTH–ICS/TR–228, Institute of Computer Science – FORTH, 1998. Available at http://www.ics.forth.gr/TR.

[LR94] F. Leymann and D. Roller. "Business Process Management with FlowMark". In *Proc. 39th IEEE Computer Society Int'l Conf. (CompCon)*, pages 230––234, 1994.

[Ltd99] WAP Forum Ltd. "WAP: Wireless Internet Today", 1999. White paper, available via WWW at http://www.wapforum.org/what/whitepapers.htm.

[Lud98] H. Ludwig. "Analysis Framework of Complex Service Performance for Electronic Commerce". In *Proc. Int'l Workshop on Business Process Reengineering and Supporting Technologies for Electronic Commerce (in conjunction with DEXA'98)*, pages 638––643, 1998.

[Lud99] H. Ludwig. "Termination Handling in Inter–Organizational Workflows : An Exception Management Approach". In *Proc. Euromicro Workshop on Parallel and Distributed Processing*, 1999.

[LW99] H. Ludwig and K. Whittingham. "Virtual Enterprise Coordinator: Agreement–Driven Gateways for Cross–Organizational Workflow Management". In *Proc. Int'l Conf. Work Activity Coordination and Collaboration (WACC)*, pages 29––38, 1999. Also Available as IBM Research Report RZ 3082, IBM Research Division, Zurich Research Laboratory, 1998.

[McB96] D. McBride. "The SLA Cookbook: A Recipe for Understanding System and Network Resource Demands". Hewlett–Packard Company, 1996. Available via URL http://www.hp.com/openview/rpm.

[MMWFF92] R. Medina–Mora, T. Winograd, R. Flores, and F. Flores. "The Action Workflow Approach to Workflow Management Technology". In *Proc. ACM Conf. on Computer–Supported Collaborative Work*, pages 281––288, 1992.

[Moh97] C. Mohan. "Recent Trends in Workflow Management Products, Standards and Research". In *Proc. NATO Advanced Institute (ASI) Workshop on Workflow Management Systems and Interoperability*, 1997. Available via URL http://www.almaden.ibm.com/cs/exotica.

[MPN97] M. Marazakis, D. Papadakis, and C. Nikolaou. "The Aurora Architecture for Developing Network−Centric Applications by Dynamic Composition of Services". Technical Report TR 213, FORTH/ICS, 1997.

[MPN98a] M. Marazakis, D. Papadakis, and C. Nikolaou. "Aurora: An Architecture for Dynamic and Adaptive Work Sessions in Open Environments". In *Proc. Int'l Conference on Database and Expert Systems Applications (DEXA'98), IEEE Computer Society Press*, pages 480−−491, 1998.

[MPN98b] M. Marazakis, D. Papadakis, and C. Nikolaou. "Management of Work Sessions in Dynamic Open Environments". In *Proc. Int'l Workshop on Workflow Management (in conjunction with DEXA'98)*, pages 725−−730, 1998.

[MPN98c] M. Marazakis, D. Papadakis, and C. Nikolaou. "The HERMES Language for Work Session Specification". In *Proc. Int'l Workshop on Coordination Technologies for Information Systems (in conjunction with DEXA'98), IEEE Computer Society Press*, pages 542−−547, 1998.

[MPN99] M. Marazakis, D. Papadakis, and C. Nikolaou. "System−level Infrastructure Issues for Controlled Interactions among Autonomous Participants in Electronic Commerce Processes". In *Proc. Int'l Workshop on Information Technologies for Electronic Commerce (in conjunction with DEXA'99), IEEE Computer Society Press*, 1999.

[MPP98] M. Marazakis, D. Papadakis, and S.A. Papadakis. "A Framework for the Encapsulation of Value−Added Services in Digital Objects". In *Proc. European Conference on Research and Advanced Technology for Digital Libraries*, pages 75−−94, 1998.

[MS93] D.R. McCarthy and S.K. Sarin. "Workflow and Transactions in InConcert". *Data Engineering Bulletin*, 16(2), 1993.

[NFC92] C. Nikolaou, D. Ferguson, and P. Constantopoulos. "Towards Goal Oriented Resource Management", 1992. IBM Research Report RC17919.

[NMP^{+}97] C. Nikolaou, M. Marazakis, D. Papadakis, Y. Yeorgiannakis, and J. Sairamesh. "Towards a Common Infrastructure to Support Large−Scale Distributed Applications". In *Proc. European Conference on Research and Advanced Technology for Digital Libraries*, pages 173−−193, 1997.

[Noo89] J. Noonan. "Automated Service Level Management and its Supporting Technologies". *Mainframe Journal*, October 1989.

[NTdS91] O. Nierstrasz, D. Tsichritzis, V. deMey, and M. Stadelmann. "Object + Scripts = Applications". In D. Tsichritzis, editor, *Object Composition*. University of Geneva, 1991.

[Ous94] J. Ousterhout. *"Tcl and the Tk Toolkit"*. Addison–Wesley, 1994.

[PCGM+96] A. Paepcke, S. B. Cousins, H. Garcia–Molina, S. F. Hassan, S. P. Ketchpel, M. Roscheisen, and T. Winograd. "Using Distributed Objects for Digital Library Interoperability". *IEEE Computer*, 29(5), 1996.

[PPC97a] S. Paul, E. Park, and J. Chaar. "Essential Requirements for a Workflow Standard". In *Proc. OOPSLA'97, Business Object Workshop III*, 1997.

[PPC97b] S. Paul, E. Park, and J. Chaar. "RainMan: A Workflow System for the Internet". In *Proc. USENIX Symposium on Internet Technologies and Systems*, 1997.

[RW97] M. Roscheisen and T. Winograd. "The FIRM Framework for Interoperable Rights Management: Defining a Rights Management Service Layer for the Internet". In *Forum on Technology–based Intellectual Property Management*, 1997. Available via URL http://pcd.stanford.edu/rmr/commpacts.html.

[Sch93] F. Schwenreis. "APRICOTS: A Prototype Implementation of a ConTract System". In *Proc. 12th IEEE Symposium on Reliable Distributed Systems*, 1993.

[Sch99] M.T. Schmidt. "The Evolution of Workflow Standards". *IEEE Concurrency*, 7(3), 1999.

[SES+97] B. Schroeder, G. Eisenhauer, K. Schwan, F. Alyea, J. Heiner, V. Martin, B. Ribarsky, M. Trauner, J. Vetter, R. Wang, and S. Zou. "Framework for Collaborative Steering of Scientific Applications". *Science Information Systems Newsletter*, IV(40), 1997.

[SGJ+96] A. Sheth, D. Georgakopoulos, S. Joosten, Rusinkiewicz, W. Scacchi, J. Wilden, and A. Wolf. Report from the NSF Workshop on Workflow and Process Automation in Information Systems. Technical report, University of Georgia, 1996. Available via URL http://LSDIS.cs.uga.edu/publications.

[She97] A. Sheth. "From Contemporary Workflow Process Automation to Adaptive and Dynamic Work Activity Coordination and Collaboration". In *Proc. Workshop on Workflow Management in Scientific and Engineering Applications*, 1997.

[SKM+96] A. Sheth, K. Kochut, J. Miller, D. Worah, S. Das, C. Lin, D. Palaniswami, J. Lynch, and I. Shevchenko. "Supporting State–Wide Immunization Tracking using Multi–Paradigm Workflow Technology". In *Proc. VLDB Conference*, 1996.

[SL97] M. Stefik and J. Lavendel. "Libraries and Digital Property Rights". In *Proc. European Conference on Advanced Technologies for Digital Libraries*, 1997.

[Swe98] K. Swenson. "Simple Workflow Access Protocol (SWAP)", 1998. Internet Draft <draft–ietf–swenson–swap–prot–00.txt>, available via WWW at URL http://www.ics.uci.edu/ ietfswap/.

[Sza97] N. Szabo. “Formalizing and Securing Relationships on Public Networks”. *First Monday*, 2(9), 1997. Available via URL http://www.firstmonday.dk/.

[T. 94] T. Berners−Lee and R. Cailliau and A. Luotonen and H. Frystyk−Nielsen and A. Secret. “The World Wide Web”. *Communications of the ACM*, 37(8):76−−82, August 1994.

[Ude] J. Udell. “Measuring Web Mindshare”. Article in Byte.com, March 1999, available via WWW at URL http://www.byte.com/features/1999/03/udellmindshare.html.

[UMA97] “Systems Management: Universal Measurement Architecture”, 1997. The Open Group, Document No. C427.

[VP98] E.M. Verharen and M.P. Papazoglou. “Introducing Contracting in Distributed Transactional Workflows”. In *Proc. 31st Annual Hawaii Int’l Conf. on System Sciences (vol. 7: Software Technology)*, pages 324−−333, 1998.

[Wei95] G. Weikum. “Workflow Monitoring: Queries on Logs or Temporal Databases ?”. In *Proc. High Performance Transaction Processing (HTPS) Workshop*, 1995.

[WfX00] “Workflow Management Coalition Workflow Standard − Interoperability Wf−XML Binding”, 2000. Document Number WFMC−TC−1023, Draft 1.0 (Beta Status) − available via WWW at URL http://www.wfmc.org.

[WR92] H. Waechter and A. Reuter. “The ConTract Model”. In *Database Transaction Models for Advanced Applications*. Morgan−Kaufman, 1992.

[Zik97] M. Zikos. “An Agent−Based Architecture for the Support of Distrubuted Information Processing Services”, 1997. MSc Thesis (in Greek).

Appendix A: IDL Specifications for the Aurora Component Framework

This appendix presents the main interfaces of the *Aurora* component framework, expressed in the OMG Interface Definition Language (IDL). The specifications rely on basic support services, such as the Query Collection, Property Set, and Messaging Framework services, the specifications of which which are omitted in the interests of brevity (along with the definitions of auxilliary data types and exception types).

Source code of the Aurora is available at http://atlas.csd.uoc.gr/aurora

```
/*
 * Component.idl -- IDL specification of a 'component', i.e. a named managed
 * object that represents a set of 'features' (in the form of functional
 * interfaces) supported by encapsulated arbitrarily complex software.
 */

module AuroraComponentModel {

   /* interface navigation-related data type definitions */
   struct ProvidedInterface {
      InterfaceName ifName; /* distinguishing name of interface */
      RepositoryID ifID; /* allows lookup in descriptor repository */
      CORBA::Object ifRef; /* could reference ComponentBase (DSI) */
   };
   struct ProvidedInterface {
      SourceName srcN;
      MessagingFramework::SubjectGroup notification_grp;
      CosQueryCollection::NamedCollection cb_collection;
   };

   /* connection management-related data type definitions */
   struct ConnectedTarget {
      ConnectionLabel label;
      RepositoryID ifID;
      CORBA::Object ref;
   };

   /* Configurator -- interface for setting configuration properties */
   interface Configurator {
      CosPropertyService::PropertySet get_configuration_properties();
      void set_config_property(in CosPropertyService::Property cfgval)
         raises(ConfiguratorException);
      void set_config_properties(in CosPropertyService::Properties cfgvals)
         raises(ConfiguratorException);
      boolean is_configuration_complete();
      void configuration_complete();
   };
```

```
/* Exporter –– navigation among the (functional) interfaces
* provided by a component. A component’s implementations is the
* ‘aggregation’ of the implementation of all the interfaces it exports.
*/
interface Exporter {
  ProvidedInterface get_provided_interface(in InterfaceName ifn)
    raises(ExporterException);
  ProvidedInterfaces get_provided_interfaces(in InterfaceNames ifns)
    raises(ExporterException);
  ProvidedInterfaces get_all_provided_interfaces()
    raises(ExporterException);
  InterfaceNameList get_provided_interface_names()
    raises(ExporterException);
  ProvidedInterface provide_interface(in InterfaceName ifn,
    in RepositoryID ifID, in CORBA::Object ifRef)
        raises(ExporterException);
  ProvidedInterfaces provide_interfaces(in InterfaceNames ifns,
    in RepositoryIDs ifIDs, in ObjectRefs ifRefs)
        raises(ExporterException);
  void revoke_provided_interface(in InterfaceName ifn)
    raises(ExporterException);
  void revoke_provided_interfaces(in InterfaceNames ifns)
    raises(ExporterException);
  void revoke_all_provided_interfaces()
    raises(ExporterException);
};

/* Importer –– management of ’connections’ between component
* interfaces. Each link associates a distinctive ’label’ to a target
* object that implements a named interface. Such links enable a
* component to ’use’ external services (provides via interfaces of
* other components). For each named interface that a component uses,
* there can be one or more ’connected’ target objects, each with its own label.
*/
interface Importer {
  void connect(in InterfaceName ifn, in ConnectionLabel lbl,
    in CORBA::Object ref)
        raises(ImporterException);
  void disconnect(in InterfaceName ifn, in ConnectionLabel lbl)
    raises(ImporterException);
  void disconnect_all(in InterfaceName ifn)
    raises(ImporterException);
  ConnectedTarget get_connected(in InterfaceName ifn, in ConnectionLabel lbl)
        raises(ImporterException);
  ConnectionLabelList get_all_connection_labels(in InterfaceName ifn)
    raises(ImporterException);
  ConnectionTargetDescription get_all_connection_targets( in InterfaceName ifn)
        raises(ImporterException);
};

/* Reactor –– interface allowing reception of event notifications
* This interface must be provided by the implementation of components.
*/
interface Reactor: MessagingFramework::CallbackObject {

  /* represents a ‘subscriber’ that exports a method with the following
  * signature: oneway void notify(in MessagingFramework::MsgReceived evM);
  */
  readonly attribute SourceName source_name;
  readonly attribute SinkName sink_name;
  readonly attribute MessagingFramework::Channel comm_channel;
```

```
    attribute MessagingFramework::CallbackHandle cb_handle;
};

/* Notifier -- management of event sources and sinks. For each
* event type, components interested in receiving notifications
* are expected to register a Reactor object. Each component can
* retrieve its 'listener' object for each specific event type (as
* identified by the 'sink' name). Each component can provide multiple
* event sources, each with multiple 'listeners'.
*/
interface Notifier {

    /* add_sink: subscribe to an event type (subject) */
    void add_sink(in SourceName src, in SinkName sn, in Reactor sink)
        raises(NotifierException);

    /* get_sink: returns the Reactor object for a named sink */
    Reactor get_sink(in SourceName src, in SinkName sn)
        raises(NotifierException);

    /* remove_sink: remove a sink from source's subscriber list */
    void remove_sink(in SourceName src, in SinkName sn)
        raises(NotifierException);

    /* remove_all_sinks: remove all sinks from subscriber list */
    void remove_all_sinks(in SourceName src)
        raises(NotifierException);

    /* get_all_source_names(): returns list of available 'subjects' */
    SourceNameList get_all_source_names()
        raises(NotifierException);

    /* get_sinks: returns all subscribers to event type */
    EventListenerList get_sinks(in SourceName src)
        raises(NotifierException);

    /* push_event: disseminate event data (name/value pairs) to subscribers */
    void push_event(in SourceName src, in Event ev)
        raises(NotifierException);
};

/* ComponentBase -- basic attributes and properties of component, plus
* accessors for Home, ConfigurationBase, Control, RequestManagement,
* EventManagement, ConnectionManagement, InterfaceNavigation objects
* associated with the component.
*/
interface ComponentBase {

    /* component name, type tag, key, and short description */
    readonly attribute string componentName;
    readonly attribute string componentKey;
    readonly attribute string componentTypeTag;
    readonly attribute string componentDescription;

    /* properties manipulated at deployment-time, including IOR's of used
    * interfaces and IOR's of event emmiters
    */
    attribute CosPropertyService::PropertySet configProps;

    /* properties manipulated at run-time, including IOR's of used
    * interfaces and IOR's of event emmiters
```

```
        */
        attribute CosPropertyService::PropertySet runtimeProps;

        /* get_descriptor() -- component descriptor object */
        AuroraComponentDescriptor::DescriptorStruct get_descriptor();

        /* get_configurator() -- configuration management object */
        Configurator get_configurator();

        /* get_control() -- monitoring/control object */
        ContainerFramework::ComponentControl get_control();

        /* get_request_manager() -- request management object */
        ContainerFramework::WorkRequestManager get_request_management_interface();

        /* get_connection_manager() -- connection management object */
        Importer get_connection_management_interface();

        /* get_interface_navigator() -- interface directory object */
        Exporter get_interface_navigator();

        /* get_event_manager() -- event dissemination management object */
        Notifier get_event_manager();
    };
};
```

Appendix B: IDL Specifications for the Aurora Container Framework

This appendix presents the main interfaces of the *Aurora* container framework, expressed in the OMG Interface Definition Language (IDL). The specifications rely on basic support services, such as the Query Collection, Property Set, Binary Data, and Security Framework services, the specifications of which which are omitted in the interests of brevity (along with the definitions of auxilliary data types and exception types).

```
/*
 * Container.idl -- IDL specification for the Aurora container framework
 * that allows clients to load, activate and monitor software components
 * within a controlled run-time environment.
 *
 */

module ContainerFramework {
   typedef sequence<octet> OctetString;
   typedef OctetString Identity;
   typedef OctetString Authority;
   struct Name {
      Identity identity;
      Authority authority;
   };
   typedef sequence<name> NameList;
   struct ClassName { /* OMG MASIF-style definition of class identity */
      string name; /* fully qualified class name */
      OctetString secID; /* class binary ID - may be used for security */
   };
   interface ComponentClassProvider {
      readonly attribute string provider_name;
      readonly attribute OctetString provider_authority;
      AuroraComponentDescriptor::ByteArrayHolder fetch_class(in ClassName classN)
         raises(ClassUnknown, ComponentClassProviderException);
      AuroraComponentDescriptor::ByteArrayHolder fetch_classes(in ClassNameList classNL)
         raises(MultipleClassesUnknown, ComponentClassProviderException);
   };
   struct MetadataDescriptor{
      string name;
      boolean is_op;
      string signature;
      string description;
   };
   typedef string ComponentStatus;
   typedef sequence<MetadataDescriptor> MetadataDescriptors;
   typedef CosPropertyService::PropertySet ComponentState;

   /* ComponentControl: uniform monitoring and control. This interface must be
```

```
* implemented by the component's developer so as to support uniform
* monitoring and control. This allows containers to export a generic
* interface for monitoring and control, by delegating requests for
* state/metadata inspection and manipulation, component−specific
* instrumentation control and communication channel manipulation to
* a ComponentControl object provided by the component's developer.
*/
interface ComponentControl {

  /* operations to start/stop/suspend/resume component execution */
  void Start(in ComponentState ctx)
    raises(ComponentControlException, UnsupportedControlOperation);
  void Stop()
    raises(ComponentControlException, UnsupportedControlOperation);
  void Suspend()
    raises(ComponentControlException, UnsupportedControlOperation);
  void Resume(in ComponentState ctx)
    raises(ComponentControlException, UnsupportedControlOperation);

  /* operation to inspect 'status' of component (active, suspended, etc) */
  ComponentStatus QueryStatus()
    raises(ComponentControlException, UnsupportedControlOperation);

  /* operations for controlling component instrumentation */
  void InitInstrumentation(in ComponentState ctx)
    raises(ComponentControlException, UnsupportedControlOperation);
  void EnableInstrumentation(in ComponentState ctx)
    raises(ComponentControlException, UnsupportedControlOperation);
  void DisableInstrumentation()
    raises(ComponentControlException, UnsupportedControlOperation);
  void StopInstrumentation()
    raises(ComponentControlException, UnsupportedControlOperation);

  /* operations for listing and manipulating component−related metadata */
  MetadataDescriptors QueryMetadata()
    raises(ComponentControlException, UnsupportedControlOperation);
  void SetMetadata(in MetadataDescriptors attrs)
    raises(ComponentControlException, UnsupportedControlOperation);
  MetadataDescriptors ListStateVars()
    raises(ComponentControlException, UnsupportedControlOperation);
  MetadataDescriptors ListControlOps()
    raises(ComponentControlException, UnsupportedControlOperation);

  /* operations for retrieving and (re)setting component state */
  ComponentState RetrieveState(in MetadataDescriptors attrs)
    raises(ComponentControlException, UnsupportedControlOperation);
  void SetState(in MetadataDescriptors attrs, in ComponentState state)
    raises(ComponentControlException, UnsupportedControlOperation);

  /* operations to establish/destroy communication channels */
  ChannelID EstablishChannel(in SubjectID subject, in ComponentInstanceKey target)
    raises(ComponentControlException, UnsupportedControlOperation);
  ChannelID DestroyChannel(in ChannelID chid)
    raises(ComponentControlException, UnsupportedControlOperation);
};

/* ComponentProxy: i/f that must be implemented by components
* hosted within a Container's run−time environment
*/
interface ComponentProxy {
  string server_name();
```

```
    ContainerID container_key();
    ComponentPackageKey component_package_key();
    CORBA::Object component_main_ref();
    ComponentControl component_control();
};

/* UpdatableComponentProxy: i/f that must be implemented by ComponentProxy
* objects that insulate customers from changes of the references to
* the actual implementations of services and their corresponding control
* objects. The callbacks specified by this i/f allow a service provider
* (or an independent third−party, such as the authority that manages the
* container run−time environment) to update or invalidate an exported
* reference, or to export alternative references that may be used by
* customers via the container's request management API. Such
* notifications allow the container to keep its map of component
* instances up−to−date, and provide a crucial building block for robust
* and scalable operation of the infrastructure in the face of
* long−duration workflow processes. Support for alternative references
* for a service allows the container to coordinate load sharing and
* fail−over without the explicit cooperation of customers, who can
* request such functionality by setting appropriate attributes in the
* service−level specifications that accompany their requests.
*/
interface UpdatableComponentProxy: ComponentProxy {
    void reset_component_main_ref(in CORBA::Object newRef);
    void alternative_component_main_ref(in CORBA::Object altRef);
    void invalidate_component_main_ref(in CORBA::Object ref);
    void invalidate_ll_component_main_refs();
    void reset_component_control(in ComponentControl newRef);
    void invalidate_component_control();
};

/* PoolObject: i/f that must be implemented by ComponentProxy objects
* that can be pooled in a LRU pool maintained by the container
*/
interface PoolObject: UpdatableComponentProxy {
    AuroraComponentDescriptor::ByteArrayHolder object_to_bytes()
       raises(PoolObjectException);
    void bytes_to_object(in OctetString bdata)
       raises(PoolObjectException);
    void object_init(in BinaryDataServiceModule::bdataID persistentRef)
       raises(PoolObjectException);
    void objectIn(in BinaryDataServiceModule::bdataID persistentRef)
       raises(PoolObjectException);
    void objectOut(in BinaryDataServiceModule::bdataID persistentRef)
       raises(PoolObjectException);
    void object_released( in BinaryDataServiceModule::bdataID persistentRef)
       raises(PoolObjectException);
    string to_string()
       raises(PoolObjectException);
};

/* Container: run−time environment for components (loader + support services)
* A container allows clients to load 'component packages', and then
* instantiate them. A client can invoke inspection and control methods
* on a component instance, as well as invoke services (possibly remote).
* It is possible to interconnect component instances, by having a
* component subscribe as listener to a subject provided by a source
* component. Publishing on a subject results in invocations of callback
* methods on all listener objects.
*/
```

```
struct ComponentPackageDescriptor {
   Name package_name;
   unsigned long access_type;
   ComponentPackageKey package_key;
   ClassNameList class_names;
   CosPropertyService::PropertySet package_properties;
};
struct ComponentInstanceDescriptor {
   ComponentPackageKey package_key;
   ComponentInstanceKey instance_key;
   Container containerRef;
   CORBA::Object component_main_objRef;
   ComponentControl component_control;
   CosPropertyService::PropertySet instance_properties;
};
interface Container {
   readonly attribute string server_name;
   readonly attribute ContainerID container_key;
   readonly attribute SecurityFramework::SecurityCredentials container_credentials;
   readonly attribute boolean supports_workflow;
   readonly attribute CosPropertyService::PropertySet serviceRefs;
   ComponentPackageKey load_component_package_drep(
            in SecurityFramework::SecurityCredentials sc,
            in string pkg_name, in string drepK, in unsigned long access_type,
            in CosPropertyService::PropertySet props)
      raises(ComponentPackageException);
   ComponentPackageKey load_component_package( in SecurityFramework::SecurityCredentials sc,
            in unsigned long access_type, in Name packageN, in ClassNameList cnames,
            in CosPropertyService::PropertySet props)
      raises(ComponentPackageException);
   void unload_component_package( in SecurityFramework::SecurityCredentials sc,
            in ComponentPackagekey pkgK)
      raises(ComponentPackageException);
   ComponentPackageKeys list_component_packages( in SecurityFramework::SecurityCredentials sc)
      raises(ComponentPackageException);
   ComponentPackageDescriptor component_package_descriptor(
            in SecurityFramework::SecurityCredentials sc, in ComponentPackageKey pkgK)
      raises(ComponentPackageException);
   ComponentInstanceKey init_component_instance( in SecurityFramework::SecurityCredentials sc,
            in ComponentPackagekey pK, in CosPropertyService::PropertySet props)
      raises(ComponentInstanceException);
   void destroy_component_instance( in SecurityFramework::SecurityCredentials sc,
            in ComponentInstanceKey cK)
      raises(ComponentInstanceException);
   ComponentInstanceKeys list_component_instances( in SecurityFramework::SecurityCredentials sc)
      raises(ComponentInstanceException);
   void reset_component_main_ref(
            in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey cK,
            in CORBA::Object newRef)
      raises(ComponentInstanceException);
   void alternative_component_main_ref(
            in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey cK,
            in CORBA::Object newRef)
      raises(ComponentInstanceException);
   void invalidate_component_main_ref(
            in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey cK,
            in CORBA::Object newRef)
      raises(ComponentInstanceException);
   void invalidate_all_component_main_refs(
            in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey cK)
      raises(ComponentInstanceException);
```

```
void reset_component_control(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey cK,
        in ComponentControl cRef)
   raises(ComponentInstanceException);
void invalidate_component_control(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey cK)
   raises(ComponentInstanceException);
void init_instrumentation(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey cK,
        in CosPropertyService::PropertySet props)
   raises(ComponentInstanceException);
void enable_instrumentation(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey cK,
        in CosPropertyService::PropertySet props)
   raises(ComponentInstanceException);
void disable_instrumentation(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey cK)
   raises(ComponentInstanceException);
void stop_instrumentation(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey cK)
   raises(ComponentInstanceException);
ComponentStatus get_status(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey cK)
   raises(ComponentInstanceException);
stringList list_control_variables(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey cK)
   raises(ComponentInstanceException);
stringList list_control_operations(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey cK)
   raises(ComponentInstanceException);
stringList get_component_instance_metadata(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey cK)
   raises(ComponentInstanceException);
void set_component_instance_metadata(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey cK, in stringList attrs)
   raises(ComponentInstanceException);
CosPropertyService::PropertySet retrieve_state(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey cK,
        in stringList state_vars, in CosPropertyService::PropertySet s)
   raises(ComponentInstanceException);
void set_state(in SecurityFramework::SecurityCredentials sc,
        in ComponentInstanceKey cK, in stringList state_vars,
        in ComponentState s)
   raises(ComponentInstanceException);
void start_component_instance(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstance cK,
        in ComponentState ctx)
   raises(ComponentInstanceException);
void suspend_component_instance(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstance cK)
   raises(ComponentInstanceException);
void resume_component_instance(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstance cK,
        in ComponentState ctx)
   raises(ComponentInstanceException);
void stop_component_instance(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstance cK)
   raises(ComponentInstanceException);
ChannelID establish_channel(
        in SecurityFramework::SecurityCredentials sc, in ComponentInstanceKey fromK,
        in SubjectID subject, in ComponentInstanceKey toK)
```

```
      raises(ComponentInstanceException);
    void destroy_channel(in SecurityFramework::SecurityCredentials sc,
            in ComponentInstance cK, in ChannelID chid)
      raises(ComponentInstanceException);
    anyList execute_method(in ComponentInstanceKey cK,
            in CallerID caller, in string if_name, in SLAspec sla,
            in string method_name, in anyList args)
      raises(ComponentInstanceException);
};

/* CallerCB: callback i/f allowing a client to receive notifications about
* whether a request that he issued has been accepted or refused, and
* also to receive the results (complete or partial) or abort notification
* about a request. All callback methods provide the RequestID of the
* request in question, since a caller may have multiple outstanding
* requests on different component instances, hosted by the same or by
* different containers.
*/
interface CallerCB {
    void request_ready_to_start(in RequestID rid, in CosPropertyService::PropertySet ctx)
      raises(WorkRequestManagerException);
    void request_refused(in RequestID rid, in CosPropertyService::PropertySet ctx)
      raises(WorkRequestManagerException);
    void request_accepted(in RequestID rid, in CosPropertyService::PropertySet ctx)
      raises(WorkRequestManagerException);
    void request_in_progress(in RequestID rid, in CosPropertyService::PropertySet ctx)
      raises(WorkRequestManagerException);
    void request_completed(in RequestID rid, in CosPropertyService::PropertySet ctx)
      raises(WorkRequestManagerException);
    void request_aborted(in RequestID rid, in CosPropertyService::PropertySet ctx)
      raises(WorkRequestManagerException);
    void request_delagated(in RequestID rid, in RequestID new_rid, in Container nc)
      raises(WorkRequestManagerException);
};

/* WorkRequestManager: specialization of Container that supports
* asynchronous request management (basis for dynamic workflow)
*
* generic interface for request management: allows application servers to
* maintain a pool of pending requests for each component, while allowing
* each component to implement its own admission control policy.
* Alternatively, a container may enforce its own policy for 'work item'
* allocation, as 'legacy' components may not be 'workflow-enabled'.
*/
interface WorkRequestManager: Container, CallerCB {
    RequestID request_init(in SecurityFramework::SecurityCredentials sc,
            in CallerID caller, in CallerCBSpec cb_spec, in ComponentInstancekey ck,
            in string ifN, in SLAspec sla, in string method, in anyList args)
      raises(WorkRequestManagerException);
    void request_start(in SecurityFramework::SecurityCredentials sc,
            in RequestID rid)
      raises(WorkRequestManagerException);
    RequestID request_issue(in SecurityFramework::SecurityCredentials sc,
            in CallerID caller, in CallerCBSpec cb_spec, in ComponentInstancekey ck,
            in string ifN, in SLAspec sla, in string method, in anyList args)
      raises(WorkRequestManagerException);
    void request_cancel(in SecurityFramework::SecurityCredentials sc,
            in RequestID rid)
      raises(WorkRequestManagerException);
    anyList request_wait(in SecurityFramework::SecurityCredentials sc,
            in RequestID rid)
      raises(WorkRequestManagerException);
    anyList request_results(in SecurityFramework::SecurityCredentials sc,
            in RequestID rid)
      raises(WorkRequestManagerException);
    RequestID request_fork(in SecurityFramework::SecurityCredentials sc,
            in RequestIDs rids)
      raises(WorkRequestManagerException);
    RequestID request_join(in SecurityFramework::SecurityCredentials sc,
            in RequestIDs rids, in unsigned long min_cnt)
      raises(WorkRequestManagerException);
    void request_with_compensation(
            in SecurityFramework::SecurityCredentials sc, in RequestID rid,
            in RequestID compensation_rid)
      raises(WorkRequestManagerException);
    void request_with_time_limit(
            in SecurityFramework::SecurityCredentials sc, in RequestID rid,
            in TimeSpec timeLimit)
      raises(WorkRequestManagerException);
    void set_request_with_time_limit(
            in SecurityFramework::SecurityCredentials sc, in RequestID rid,
            in TimeSpec timeLimit)
      raises(WorkRequestManagerException);
    RequestIDs list_pending_requests( in SecurityFramework::SecurityCredentials sc)
      raises(WorkRequestManagerException);
};

/* ContainerServer: factory/directory for Container objects
*/
interface ContainerServer {
    readonly attribute string server_name;
    readonly attribute SecurityFramework::SecurityCredentials server_sc;
    ContainerID create_container( in SecurityFramework::SecurityCredentials sc, in string name,
      in boolean supports_workflow, in unsigned long max_active_instances,
      in unsigned long long expiration_timeout)
            raises(ContainerServerException);
    void shutdown_container( in SecurityFramework::SecurityCredentials sc, in ContainerID cid)
      raises(ContainerServerException);
    ContainerIDs list_containers( in SecurityFramework::SecurityCredentials sc)
      raises(ContainerServerException);
    Container get_container(in ContainerID cid)
      raises(ContainerServerException);
};
};
```

Appendix C: IDL Specifications for the Aurora Service−Level Agreement Framework

This appendix presents the main interfaces of the *Aurora* service−level agreement framework, expressed in the OMG Interface Definition Language (IDL). The specifications rely on basic support services, such as the Query Collection, Property Set, Binary Data, Security Framework, and Logging/Monitoring services, the specifications of which which are omitted in the interests of brevity (along with the definitions of auxilliary data types and exception types).

```
/*
 * SLA.idl −− IDL specification for a service−level agreement framework
 *
 * − Service−level agreements encapsulate the context of an interaction
 * between a service provider and a client. As an example, a SLA can
 * coordinate the interaction between a merchant and a customer in the
 * case of an electronic commerce transaction. The SLA is executed either
 * by the service provider or by a mutually trusted thrid−party, in order
 * to coordinate the actions of the participants. In the e−commerce example,
 * the SLA could specify what the participants (merchant, customer, banks,
 * etc) should do next in the way of ordering, payment, and delivery.
 *
 * − All participants in an interaction, including the SLA object, respond to
 * incoming requests by taking local actions (eg: authorize a payment,
 * initiate a delivery process) and issuing requests to other participants
 * so as to advance the state of the ongoing interaction.
 *
 * − The event handlers in each of the participants can be 'proxies' to
 * autonomous systems, thus insulating the SLA as well as the participants
 * from the implementation details for particular actions.
 *
 * − SLA objects are 'generic' in the sense that the details of implementing
 * a specific 'contract' are contained exclusively within the provider's
 * event handlers.
 *
 * − The interface declarations in this specification are intended to serve
 * as 'inheritance base' for specific process implementations (eg: an
 * e−commerce transaction scenario).
 *
 */

module SLAframework {

   /* auxilliary data types */
   typedef CospropertyService::PropertySet SLAstate;
```

```
struct SLAevent {
   TimeSpec ts;
   StatusTag status;
   SLAstate data;
};
typedef AuroraComponentDescriptor::Resourcekey ServiceDescriptor;
typedef ContainerFramework::ComponentInstanceKey ServiceHandle;
typedef CosQueryCollection::Collection ActionCollection;
typedef CosQueryCollection::Iterator ActionIterator;
typedef CosQueryCollection::Iterator HistoryIterator;
typedef any TermsAndConditions;
typedef string StatusTag;

/* SLA */
interface SLA: SecurityFramework::Actor {
   readonly attribute string description;
   readonly attribute SecurityFramework::SecurityCredentials authority;
   SLAid sla_id();
   LoggingServiceModule::SessionID session_id();
   LoggingServiceModule::TaskID task_id();
   void set_status(in SecurityFramework::SecurityDCredentials sc,
            in StatusTag status)
      raises(SLAexception);
   StatusTag get_status(in SecurityFramework::SecurityCredentials sc)
      raises(SLAexception);
   void set_sla_state(in SecurityFramework::SecurityCredentials sc,
            in SLAstate state)
      raises(SLAexception);
   SLastate get_sla_state(in SecurityFramework::SecurityCredentials sc)
      raises(SLAexception);
   void start(in SecurityFramework::SecurityCredentials sc,
            in StatusTag status, in SLAstate state)
      raises(SLAexception);
   void terminate(in SecurityFramework::SecurityCredentials sc,
            in StatusTag status)
      raises(SLAexception);
   void complete(in SecurityFramework::SecurityCredentials sc)
      raises(SLAexception);
   void passivate(in SecurityFramework::SecurityCredentials sc)
      raises(SLAexception);
   void reactivate(in SecurityFramework::SecurityCredentials sc,
            in StatusTag status, in SLAstate state)
      raises(SLAexception);
   HistoryIterator get_history(in SecurityFramework::SecurityCredentials sc)
      raises(SLAexception);
   void sla_event(in SecurityFramework::SecurityCredentials sc,
            in SLAevent ev)
      raises(SLAexception);
   TermsAndConditions get_contract_specification(
            in SecurityFramework::SecurityCredentials sc)
      raises(SLAexception);
   ServiceHandle get_service_handle( in SecurityFramework::SecurityCredentials sc)
      raises(SLAexception);
   ActionIterator get_supported_actions( in SecurityFramework::SecurityCredentials sc)
      raises(SLAexception);
   ServiceDescriptor get_service_descriptor(
            in SecurityFramework::SecurityCredentials sc)
      raises(SLAexception);
   ContainerFramework::RequestID request_fulfillment(
            in SecurityFramework::SecurityCredentials sc, in SLAspec spec,
            in SecurityFramework::Action action, in SLAstate params)
      raises(SLAexception);
```

```
        void declare_fulfilled(in SecurityFramework::SecurityCredentials sc,
                  in ContainerFramework::RequestID rid, in SLAstate state)
           raises(SLAexception);
        void declare_exception(in SecurityFramework::SecurityCredentials sc,
                  in ContainerFramework::RequestID rid, in SLAstate state)
           raises(SLAexception);
        void declare_satisfied(in SecurityFramework::SecurityCredentials sc,
                  in ContainerFramework::RequestID rid)
           raises(SLAexception);
        void declare_complaint(in SecurityFramework::SecurityCredentials sc,
                  in ContainerFramework::RequestID rid, in SLAstate state)
           raises(SLAexception);
     };

     /* SLA factory */
     interface SLAfactory {
        SLA create_sla(in SecurityFramework::SecurityCredentials sc,
                  in string description, in TermsAndConditions contract,
                  in ActionCollection actions, in ServiceHandle serviceH,
                  in ServiceDescriptor descriptor_key, in LoggingServiceModule::SessionID sid,
  in LoggingServiceModule:TaskID tid,
                  in CosPropertyService:;PropertySet exportedProps,
                  in string slaName, in string slaClassName)
           raises(SLAfactoryException);
        SLA get_sla(in SecurityFramework::SecurityCredentials sc,
                  in SLAid sla_id)
           raises(SLAfactoryException);
        void terminate(in SecurityFramework::SecurityCredentials sc,
                  in SLAid sla_id, in StatusTag status)
           raises(SLAfactoryException);
        void complete(in SecurityFramework::SecurityCredentials sc,
                  in SLAid sla_id)
           raises(SLAfactoryException);
        void passivate(in SecurityFramework::SecurityCredentials sc,
                  in SLAid sla_id)
           raises(SLAfactoryException);
        void reactivate(in SecurityFramework::SecurityCredentials sc,
                  in SLAid sla_id, in StatusTag status, in SLAstate state)
           raises(SLAfactoryException);
        SLAids list_slas(in SecurityFramework::SecurityCredentials sc)
           raises(SLAfactoryException);
     };
  };
```

Appendix D: IDL Specification of the Aurora Work Session Framework

This appendix presents the main interfaces of the *Aurora* container framework, expressed in the OMG Interface Definition Language (IDL). The specifications rely on basic support services, such as the Query Collection, Property Set, Binary Data, and Security Framework services, the specifications of which which are omitted in the interests of brevity (along with the definitions of auxilliary data types and exception types).

```
/*
 * Session.idl -- IDL specification of the Aurora work session framework
 *
 * - This specification defines a model that represents basic objects and
 * their relationships in the end-user view of a distributed system. The
 * objective is to provide a consistent and common description of fundamental
 * concepts such as users, resources, tasks so as to enable applications to
 * significantly raise the level of collaboration and resource sharing within
 * projects and enterprises, while also raising the level of abstraction for
 * the end-users that directly interact with the distributed system, in the
 * context of 'work sessions', which are defiend as configurations of
 * participants in shared workspaces using resources in processes.
 */

module SessionFramework {
    typedef ResourceEvent {
        string event_type;
        any event_data;
    };
typedef SharedWorkspace Session;
    typedef sequence<string> EventTypeTags;

    * ResourceEventReceiver: i/f for receiving notifications of type

ResourceEvent (string tag (event_type) + 'any' value that contains a

MessagingFramework::ConditionNotification)

/
    interface ResourceEventReceiver {
        EventTypeTags get_supported_event_types();
        oneway void resource_event(in ResourceEvent ev);
        ResourceEventIterator get_history()
            raises(SessionFrameworkException);
    };

    /* SessionEventCallback: i/f for receiving notifications of session-related
    * events: added/removed in workspace, produced/consumed by task
    */
```

```
interface SessionEventCallback {
   void resource_added_in_workspace(in Session wpsace);
   void resource_removed_from_workspace(in Session wpsace);
   void resource_produced_by_task(in Task ptask);
   void resource_consumed_by_task(in Task ptask);
};

/* SessionResource:
* – inherits the following methods from the Actor interface:
* name(), get_credentials(), get_exported_properties(),
* set_exported_properties()
*/
interface SessionResource: SecurityFramework::Actor, SessionEventCallback {
   string rname();
   string rkey();
   string rtype();
   string rdescription();
   void set_ctx(in CosPropertyService::PropertySet ctx)
      raises(SessionFrameworkException);
   CosPropertyService::PropertySet get_ctx();
   WorkspaceIterator get_iterator_on_workspaces();
   TaskIterator get_iterator_on_producers();
   TaskIterator get_iterator_on_consumers();
   ContainerFramework::ComponentInstanceKey component_ref();
   void release(in StatusCode status)
      raises(SessionFrameworkException);
};

/* Session Participant */
interface SessionParticipant: SessionResource {
   void submit_request(in WorkRequest req)
      raises(SessionFrameworkException);
   void remove_request(in WorkRequest req)
      raises(SessionFrameworkException);
   WorkRequestIterator get_request_iterator()
      raises(SessionFrameworkException);
   Task create_task(in string name, in TaskID key,
         in SharedWorkspace work_session, in SessionResource data)
      raises(SessionFrameworkException);
   TaskIterator get_task_iterator()
      raises(SessionFrameworkException);
   SharedWorkspace workspace();
   SharedWorkspace create_workspace(in string name, in SessionID key,
         in SecurityFramework::ACL acl, in string ws_class_name)
      raises(SessionFrameworkException);
};

/* WorkRequest:
* – inherits the following methods from the Action interface:
* target(), callerRolesRequired()
*/
interface WorkRequest {
   ContainerFramework::RequestID request_id();
   any request_body();
};

/* Shared Workspace */
interface SharedWorkspace: SessionResource, ResourceEventReceiver {
   SecurityFramework::ACL get_acl();
   SharedWorkspace create_subworkspace(in string name, in WorkspaceID key,
         in string description, in SecurityFramework::ACL acl, in string ws_class_name)
      raises(SessionFrameworkException);
    void add_resource(in SessionResource r)
      raises(SessionFrameworkException);
    void remove_resource(in SessionResource r)
      raises(SessionFrameworkException);
    ResourceIterator get_resource_iterator(in string resourceType)
      raises(SessionFrameworkException);
};

/* Task */
interface Task: SessionResource, ResourceEventReceiver {
    string state();
    SessionParticipant owner();
      void set_owner(in SessionParticipant owner)
      raises(SessionFrameworkException);
    void add_consumed(in SessionResource r)
      raises(SessionFrameworkException);
    void remove_consumed(in SessionResource r)
      raises(SessionFrameworkException);
    ResourceIterator get_consumed_resource()
      raises(SessionFrameworkException);
    void add_produced(in SessionResource r)
      raises(SessionFrameworkException);
    void remove_produced(in SessionResource r)
      raises(SessionFrameworkException);
    ResourceIterator get_produced_resource()
      raises(SessionFrameworkException);
    void start()
      raises(SessionFrameworkException);
    void stop()
      raises(SessionFrameworkException);
    void suspend()
      raises(SessionFrameworkException);
    void resume()
      raises(SessionFrameworkException);
};

/* Session Manager */
interface SessionManager {
    SessionID open_session(in SecurityFramework::SecurityCredentials sc,
          in string name, in string skey, in string descr,
          in SecurityFramework::ACL acl, in CosPropertyService::PropertySet ctx)
      raises(SessionFrameworkException);
    void close_session(in SecurityFramework::SecurityCredentials sc,
          in SessionID sid, in StatusCode status)
      raises(SessionFrameworkException);
    void join_session(in SecurityFramework::SecurityCredentials sc,
          in SessionID sid, in SessionResource r)
      raises(SessionFrameworkException);
    void leave_session(in SecurityFramework::SecurityCredentials sc,
          in SessionID sid, in SessionResource r)
      raises(SessionFrameworkException);
    CosPropertyService::PropertySet get_session_state(
          in SecurityFramework::SecurityCredentials sc, in SessionID sid)
      raises(SessionFrameworkException);
    Session get_session(in SecurityFramework::SecurityCredentials sc,
          in SessionID sid)
      raises(SessionFrameworkException);
    SessionIDs list_sessions(in SecurityFramework::SecurityCredentials sc)
      raises(SessionFrameworkException);
    SessionID get_session_byName( in SecurityFramework::SecurityCredentials sc,
            in string name)
        raises(SessionFrameworkException);
      SessionID get_session_byAttributes( in SecurityFramework::SecurityCredentials sc,
            in CosPropertyService::Properties attrs)
        raises(SessionFrameworkException);
   };
};
```

Appendix E: IDL Specification of a Workflow Facility based on the OMG jointFlow Standard

This appendix presents the main interfaces of the workflow facility described in the case study of Chapter 7. This workflow facility is modeled after the OMG jointFlow specification [jFl98], with minor adjustmentments to re−use some of the basic support services of the Aurora run−time ennvironment.

```
/*
 * jFlow.idl −− IDL specification for a workflow execution and monitoring
 * infrastructure that is similar in spirit to the OMG jointFlow standard
 * as described in OMG Document Number bom/98−06−07
 */

module jFlowFramework {
    typedef string StatusCode;
    typedef string StateID;
    typedef CosQueryCollection::Iterator AssignmentMemberIterator;
    typedef CosQueryCollection::Iterator ActivityIterator;
    typedef CosQueryCollection::Iterator ProcessIterator;
    typedef CosQueryCollection::Iterator HistoryIterator;
    typedef CosPropertyService::PropertySet Data;
    typedef DataSignatureStruct {
        string attribute_name; /* name of argument/parameter */
        string type_name; /* IFR id for type of argument/parameter */
    };
    typedef sequence<DataSignatureStruct> DataSignature;

    /* Resource */
    interface Resource {
        string name();
        string key();
        string description();
        AssignmentMemberIterator get_iterator_on_assignments()
            raises(ProcessException);
        boolean is_member_of_assignment(in Assignment assignment)
            raises(ProcessException);
        void release(in Assignment assignment, in StatusCode status)
            raises(ProcessException);
    };

    /* Requester */
    interface Requester {
        ProcessIterator get_iterator_on_performers()
            raises(ProcessException);
        boolean is_associated_with_process(in WorkflowProcess process)
        raises(ProcessException);
    void notification(in EventNotification ev)
        raises(ProcessException);
};

/* Perfomer */
interface Performer {
    string name();
    string key();
    string description();
    StateID state();
    void set_state(in StateID state)
        raises(ProcessException);
    Data context_data();
    void set_context_data(in Data ctx_data)
        raises(ProcessException);
    unsigned long priority();
    void set_priority(in unsigned long prio)
        raises(ProcessException);
    void suspend()
        raises(ProcessException);
    void resume()
        raises(ProcessException);
    void abort()
        raises(ProcessException);
    void terminate()
        raises(ProcessException);
    HistoryIterator get_history_iterator()
        raises(ProcessException);
};

/* Activity */
interface Activity: Requester, Performer {
    Data result();
        raises(ProcessException);
    void set_result(in Data res)
        raises(ProcessException);
    WorkflowProcess parent_process();
    boolean is_member_of_assignment(in Assignment assignment)
        raises(ProcessException);
    AssignmentMemberIterator get_iterator_on_assignments()
        raises(ProcessException);
    void complete()
        raises(ProcessException);
};

/* Workflow Process */
interface WorkflowProcess: Performer {
    Requester process_requester();
    PrcoessManager process_mgr();
    ActivityIterator get_iterator_on_activities()
        raises(ProcessException);
    boolean is_parent_of_activity(in Activity activity)
        raises(ProcessException);
    Dataa result()
        raises(ProcessException);
    void start()
        raises(ProcessException);
};

/* Process Manager */
```

```
    interface ProcessManager {
        string name();
        string key();
        string description();
        ProcessIterator get_iterator_on_processes()
            raises(ProcessException);
        boolean is_creator_of_process(in WorkflowProcess process)
            raises(ProcessException);
        DataSignature context_data_signature();
        DataSignature result_data_signature();
        WorkflowProcess create_process(in Requester requester)
            raises(ProcessException);
    };

    /* Assignment
    * – allowed values for ‘state’: pending_approval, approved
    */
    enum assignment_state {
        no_assignment, pending_approval, approved
    };
    interface Assignment {
        string assignment_name();
        string assignment_key();
        string assignment_description();
        Activity assigned_to_activity();
        Resource assigned_resource()
            raises(ProcessException);
        void set_resource(in Resource r)
            raises(ProcessException);
        assignment_state state();
        void set_state(in assignment_state s)
            raises(ProcessException);
    };

    /* EventNotification:
    * – allowed values for ‘event_type’: create_process, process_state_change,
    * activity_state_change, activity_ctx_update, activity_result_update,
    * assignment_state_change, process_ctx_change
    */
    enum event_notification_type {
        create_process, process_state_change, activity_state_change, activity_ctx_change,
        activity_result_update, activity_ctx_change, activity_result_update,
        assignment_state_change, process_ctx_change
    };
    interface EventNotification {
        Performer source();
        string event_type();
        TimeSpec ts();
        string process_name();
        string process_key();
        string activity_name();
        string activity_key();
        string proces_mgr_name();
        string prev_state();
        string new_state();
        string prev_assignment_state();
        string new_assignment_state();
    };
};
```

Appendix F: IDL Specification for the Electronic Commerce Case Study

This appendix presents the main interfaces of the electronic commerce framework described in the case study of Chapter 7. Only the definitions of auxilliary data types and exception types have been omitted in the interests of brevity.

```
/*
 * ecommerce.idl -- IDL specification for services supporting electronic
 * commerce: products catalogue, order tracking, shopping basket,
 * payment processor, problem log
 *
 * - The implementation of these support services uses the LogMonitor service.
 */

module EcommerceModule {

   /* ProductsCatalogue: i/f of products database
   * - distinction between 'soft' and 'hard' goods: Soft goods can be
   * obtained immediately, whereas hard goods require a fulfillment
   * process before the order can be marked for shipping.
   * - Product entries cannot be deleted, since a product may have been
   * ordered thus creating an entry in the order tracking system.
   */
   struct ProductStruct {
      ProductID product_id;
      string product_name;
      string product_description;
      double price;
      string currency;
      boolean soft_good_p;
      boolean for_sale_p;
      string terms_and_conditions;
      Quantity quantity; /* defined in the context of the Inventory service */
   };
   typedef sequence<ProductStruct> ProductStructList;

   interface ProductsCatalogue: ContainerFramework::StatisticsSource {
      AuroraComponentDescriptor::ByteArrayHolder get_product_info(
               in SecurityFramework::SecurityCredentials sc, in ProductID product_id)
        raises(ProductsCatalogueException);
      AuroraComponentDescriptor::ByteArrayHolder get_current_products(
               in SecurityFramework::SecurityCredentials sc)
        raises(ProductsCatalogueException);
      AuroraComponentDescriptor::ByteArrayHolder get_discontinued_products(
               in SecurityFramework::SecurityCredentials sc)
        raises(ProductsCatalogueException);
      ProductID add_product_entry( in SecurityFramework::SecurityCredentials sc,
               in string name, in string descr, in boolean soft_good_p,
              in double price, in string currency, in string terms)
        raises(ProductsCatalogueException);
    void modify_product_entry( in SecurityFramework::SecurityCredentials sc,
              in ProductID product_id, in string name, in string descr, in boolean soft_good_p,
              in boolean for_sale_p, in double price, in string currency, in string terms)
        raises(ProductsCatalogueException);
    ProductIDs list_product_ids( in SecurityFramework::SecurityCredentials sc)
        raises(ProductsCatalogueException);
};

/* Inventory: specialization of ProductsCatalogue i/f to keep track of
* the available quantity for each product and to support automatic
* alerts when the inventory level for a product becomes higher/lower
* than a specified threshold. It is assumed that when a product is
* 'discontinued' an alert is signalled to all clients that have
* registered an interest in monitoring this product's 'quantity'.
*/

enum InventoryAlertType {
    LT_THRESHOLD, EQ_THRESHOLD, GT_THRESHOLD
};
struct InventoryLevelNotificationStruct {
    ProductID product_id;
    boolean discontinued;
    Quantity prev_quantity;
    Quantity delta;
};
interface Inventory: ProductsCatalogue, ContainerFramework::StatisticsSource {
    void update_quantity(in SecurityFramework::SecurityCredentials sc,
              in ProductID product_id, in Quantity delta)
        raises(InventoryException);
    Quantity get_quantity(in SecurityFramework::SecurityCredentials sc,
              in ProductID product_id)
        raises(InventoryException);
    InventoryAlertID request_alter( in SecurityFramework::SecurityCredentials sc,
              in ProductID product_id, in Quantity threshold, in InventoryAlertType alter_type,
              in MessagingFramework::CallbackObject cb)
        raises(InventoryException);
    void cancel_alert(in SecurityFramework::SecurityCredentials sc,
              in InventoryAlertID alter_id)
        raises(InventoryException);
};

/* OrderTracking: i/f of order management/tracking system
* – supports queries over particular orders. queries over the
* time series of orders so far (history view), as well as grouping
* orders by product, date, and/or customer name.
*/

struct OrderStruct {
    OrderID order_id;
    ProductIDs product_ids;
    Quantities quantities;
    double tot_amount;
    PaymentMethodSpecification payment_details;
    LoggingServiceModule::SessionID sid;
    LoggingServiceModule::TaskID tid;
    LoggingServiceModule::SourceIdentity srcID;
    TimeSpec confirmed_date;
    TimeSpec expiration_date;
    TimeSpec shipped_date;
```

```
        TimeSpec last_change_date;
        CustomerName cust_name;
        CustomerAddress cust_addr;
        CustomerPhone cust_phone;
        CustomerEmail cust_email;
        ShippingAddress shipping_addr;
        OrderState order_state;
        /* Possible order states: INIT, void, confirmed, failed_authorization,
        * authorized, shipped, returned, refunded, expired
        */
    };
    typedef sequence<OrderStruct> OrderStructList;
    interface OrderTracking: ContainerFramework::StatisticsSource {
        OrderID add_order(in SecurityFramework::SecurityCredentials sc,
                in ProductsIDs product_ids, in Quantities quantities, in double tot_amount,
                in LoggingServiceModule::SessionID sid, in LoggingServiceModule::TaskID tid,
                TimeSpec confirmed_date, in TimeSpec expiration_date, in TimeSpec shipped_date,
                in CustomerName cust_name, in CustomerAddress cust_addr, in CustomerPhone cust_phone,
                in CustomerEmail cust_email, in PaymentMethodSpecification payment_details,
                in ShippingAddress shipping_addr, in BankAuthorizationCode bcode)
          raises(OrderTrackingException);
        void set_order_state(in SecurityFramework::SecurityCredentials sc,
                in OrderID order_id, in OrderState order_state)
          raises(OrderTrackingException);
        OrderState get_order_state( in SecurityFramework::SecurityCredentials sc,
                in OrderID order_id)
          raises(OrderTrackingException);
        void set_shipped_date(in SecurityFramework::SecurityCredentials sc,
                in OrderID order_id, in TimeSpec shipped_date)
          raises(OrderTrackingException);
        AuroraComponentDescriptor::ByteArrayHolder get_order_info(
                in SecurityFramework::SecurityCredentials sc, in OrderID order_id)
          raises(OrderTrackingException);
        AuroraComponentDescriptor::ByteArrayHolder get_orders(
                in SecurityFramework::SecurityCredentials sc, in ProductID product_id, in OrderState order_state,
                in CustomerName cust_name, in TimeSpec startTS, in TimeSpec endTS)
          raises(OrderTrackingException);
    };

    /* ShoppingBasket: i/f for combining products from one or more
    * ProductsCatalogue services into a single order.
    *
    * – submit_order() results in issuing orders for products listed in one or
    * more ProductsCatalogue's, via a specified OrderTracking service.
    */

    typedef SB_State {
        INITIALIZED, SUBMITTED, EXPIRED
};
    struct ShoppingBasketStruct {
        ShoppingBasketID sb_id;
        SB_State sb_state;
        LoggingServiceModule::SessionID sid;
        LoggingServiceModule::TaskID tid;
        LoggingServiceModule::SourceIdentity srcID;
        TimeSpec init_change;
        TimeSpec last_change_date;
        TimeSpec expiration_date;
        TimeSpec submit_date;
        ProductIDs product_ids;
        Quantities quantities;
        OrderTrackingSvcCosName order_tracking_svc;
    };
    typedef sequence<ShoppingBasketStruct> ShoppingBasketStructList;
    struct SB_Item {
        ProductID product_id;
        Quantity quantity;
    };
    typedef sequence<SB_Item> SB_ItemList;
    interface ShoppingBasket: ContainerFramework::StatisticsSource {
        ShoppingBasketID init_sb(in SecurityFramework::SecurityCredentials sc,
                in TimeSpec expiration_date, in LoggingServiceModule::SessionID sid,
                in LoggingServiceModule::TaskID tid)
            raises(ShoppingBasketException);
        void add_item(in SecurityFramework::SecurityCredentials sc,
                in ShoppingBasketID sb_id, in ProductID product_id, in Quantity quantity_delta)
            raises(ShoppingBasketException);
        void remove_item(in SecurityFramework::SecurityCredentials sc,
                in ShoppingBasketID sb_id, in ProductID product_id, in Quantity quantity_delta)
            raises(ShoppingBasketException);
        OrderID submit_order(in SecurityFramework::SecurityCredentials sc,
                in ShoppingBasketID sb_id, in OrderTrackingSvcCosName ot_svc_cosN,
                in PaymentMethodSpecification payment_details, in CustomerName cust_name,
                in CustomerAddress cust_addr, in CustomerPhone cust_phone,
                in CustomerEmail cust_email, in ShippingAddress shipping_addr,
                in BankAuthorizationCode bcode, in TimeSpec expiration_date)
            raises(ShoppingBasketException);
        AuroraComponentDescriptor::ByteArrayHolder list_items(
                in SecurityFramework::SecurityCredentials sc, in ShoppingBasketID sb_id)
            raises(ShoppingBasketException);
        SB_State get_sb_state(in SecurityFramework::SecurityCredentials sc,
                in ShoppingBasketID sb_id)
            raises(ShoppingBasketException);
        AuroraComponentDescriptor::ByteArrayHolder get_sb(
                in SecurityFramework::SecurityCredentials sc,
in ShoppingBasketID sb_id)
            raises(ShoppingBasketException);
        void drop_sb(in SecurityFramework::SecurityCredentials sc,
                ShoppingBasketID sb_id)
            raises(ShoppingBasketException);
        ShoppingBasketIDs list_shopping_basket_ids(
                in SecurityFramework::SecurityCredentials sc)
            raises(ShoppingBasketException);
    };

    /* PaymentProcessor: minimal wrapper (front−end) for electronic payment
    * – send_command_to_server() allows using the payment processor's API
    * directly (eg: CyberCash: mauth, postauth, mauthcapture, query, etc).
    * – do_direct_payment() is a convenience method, grouping together all
    * the steps required for completing a payment transaction.
    * – get_tx_history() returns all log records related to a specified
    * task within a session thus providing a history of 'significant'
    * events during the course of a payment transaction.
    */

    interface PaymentProcessor: ContainerFramework::StatisticsSource {
        unsigned long send_command( in SecurityFramework::SecurityCredentials sc,
                in LoggingServiceModule::SessionID sid, in LoggingServiceModule::TaskID tid,
                in CosPropertyService::PropertySet ins, in CosPropertyService::PropertySet outs)
            raises(PaymentProcessorException);
        unsigned long execute_direct_payment( in SecurityFramework::SecurityCredentials sc,
                in LoggingServiceModule::SessionID sid, in LoggingServiceModule::TaskID tid,
                in CosPropertyService::PropertySet ins, in CosPropertyService::PropertySet outs)
            raises(PaymentProcessorException);
        AuroraComponentDescriptor::ByteArrayHolder get_tx_history(
                in SecurityFramework::SecurityCredentials sc,
                in LoggingServiceModule::SessionID sid, in LoggingServiceModule::TaskID tid)
            raises(PaymentProcessorException);
    };

    /* ProblemLog: service for logging problem reports so that service providers
    * can inspect them and take appropriate action to handle them
    */

    struct ProblemStruct {
        ProblemID problem_id;
        OrderID order_id;
        LoggingServiceModule::SessionID sid;
        LoggingServiceModule::TaskID tid;
        LoggingServiceModule::SourceIdentity srcID;
        TimeSpec problem_date;
        TxType tx_type;
        TxStatus tx_status;
        string err_msg;
    };
    typedef sequence<ProblemStruct> ProblemStructList;
    interface FeedbackLog: ContainerFramework::StatisticsSource {
        ProblemID add_problem_entry( in SecurityFramework::SecurityCredentials sc,
                in OrderID order_id, in TxType tx_type, in TxStatus tx_status,
                in string err_msg, in LoggingServiceModule::SessionID sid,
                LoggingServiceModule::taskID tid)
            raises(FeedbackLogException);
        void remove_problem_entry( in SecurityFramework::SecurityCredentials sc,
                in ProblemID problem_id)
            raises(FeedbackLogException);
        AuroraComponentDescriptor::ByteArrayHolder get_problem_entry(
                in SecurityFramework::SecurityCredentials sc, in ProblemID problem_id)
            raises(FeedbackLogException);
        AuroraComponentDescriptor::ByteArrayHolder get_problem_entries_on_order(
                in SecurityFramework::SecurityCredentials sc, in OrderID order_id)
            raises(FeedbackLogException);
        AuroraComponentDescriptor::ByteArrayHolder get_problem_log(
                in SecurityFramework::SecurityCredentials sc)
            raises(FeedbackLogException);
        ProblemIDs list_problem_ids( in SecurityFramework::SecurityCredentials sc)
            raises(FeedbackLogException);
    };
};
```