Structure de COOL V1

Julien Maisonneuve, H. Soulard

▶ To cite this version:

Julien Maisonneuve, H. Soulard. Structure de COOL V1. RT-0127, INRIA. 1991, pp.19. inria-00070040

HAL Id: inria-00070040
https://inria.hal.science/inria-00070040

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports Techniques

N° 127

Programme 1
Architectures Parallèles, Bases de Données, Réseaux et Systèmes

STRUCTURE DE COOL V1

Julien MAISONNEUVE
Hervé SOULARD

Mars 1991

Structure de COOL V1

COOL V1 Structure

Julien Maisonneuve
Hervé Soulard

INRIA, Projet SOR

27 février 1991

Résumé

COOL est une couche à objets au-dessus du système réparti Chorus [1]. Le modèle d'objet de COOL est de grain moyen ou gros, les objets peuvent être actifs et recevoir des messages, ou au contraire rester passifs et être invoqués dans un contexte local. Ils peuvent être assemblés en arbres par un mécanisme de membre. COOL permet la migration d'arbres d'objets, et dans le cas d'objets actifs, dans un état bien défini. Le prototype COOL est réalisé avec un simulateur Chorus au-dessus d'UNIX. Ce document cherche à donner une idée de la structure et du fonctionnement de COOL dans ce contexte.

Abstract

COOL is an object-oriented layer above the Chorus distributed system. The COOL objetc model is medium or coarse, objects can be active and receive messages, or stay passive and be invoked within a local context. They can be assembled through a member mechanism. COOL allows object trees migration, in a fixed state when they are active. The COOL prototype is built on top of a Chorus simulator, itself above UNIX. This document is an attempt to give an idea of the structure and of the inner workings of COOL in this context.

[1] Ce rapport a été réalisé dans le cadre d'un contrat de recherche entre Chorus Systèmes, le SEPT (service d'études commun des postes et télécommunications) et le projet SOR de l'INRIA.

1 Introduction

COOL est un système réparti orienté objet construit au-dessus du simulateur du noyau Chorus [2] V3. COOL est un projet commun à Chorus Systèmes et à l'INRIA d'une part, et au SEPT d'autre part [3]. Le but de ce projet était de fournir un environnement réparti orienté objet pour permettre le développement par le SEPT de l'application documentaire CIDRE [Deguin 1989].

Ce document a pour but de donner une idée de la structure interne du système COOL tel qu'il est réalisé au-dessus du simulateur UNIX. On suppose que le lecteur est familier avec les abstractions de COOL, de Chorus et bien entendu d'UNIX [Habert 1990, Habert 1989, Rozier 1988b, Rozier 1988a, Abrossimov 1989, Armand 1989]. Les fonctions proposées par le système COOL sont le fruit de la coopération de quatres couches, chacune s'appuyant sur la précédente :

- Le système UNIX au plus bas niveau,
- Le simulateur Chorus,
- Le système COOL,
- Le chargeur RUN de COOL.

2 Le Simulateur Chorus

Le simulateur Chorus est construit au-dessus d'UNIX et utilise les objets de ce système pour matérialiser ses propres abstractions.

Les acteurs Chorus sont constitués à partir des processus UNIX. Comme UNIX est un sytème mono-thread, il ne peut fournir qu'une seule activité par processus, il faut simuler le scheduling des threads à un niveau supérieur.

Dans certains cas, le système COOL lui-même est amené à utiliser UNIX, en particulier le système de fichiers qui n'existe pas dans le simulateur. Il est la base du service de nommage de COOL, le couple Chorus/COOL (tout particulièrement le Mapper) s'en sert pour construire les abstractions de segments, y compris les segments contenant les descriptions de classes et enfin COOL l'utilise pour les appels d'entrées-sorties pour ses objets.

COOL doit aussi se frotter à UNIX partout où la simulation de Chorus offerte est imparfaite.

2.1 Les librairies Chorus

[2] Chorus est une marque déposée de Chorus Systèmes

[3] Service d'Etudes commun des Postes et Télécommunications

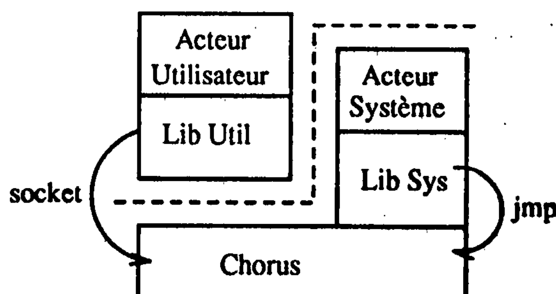

Figure 1 : Librairies Chorus

Il y a deux librairies distinctes (figure 1) contenant les appels systèmes Chorus :

L'une d'elles fournit aux acteurs utilisateurs les primitives du noyau Chorus. La communication entre les acteurs et le noyau se fait au travers d'une simulation de trappe système réalisée à base de sockets du domaine UNIX.

Les acteurs système chargés avec le noyau Chorus (tel le système COOL) communiquent avec le noyau Chorus au travers d'une bibliothèque qui simule une trappe système par un saut (jmp).

2.2 Création d'un acteur Chorus sur le simulateur UNIX

Les acteurs Chorus sont normalement créés par actorCreate(). Dans le simulateur, cet appel lance un acteur vide.

La création des acteurs se fait de la manière suivante : Lorsqu'un processus UNIX (lançé avec fork()) invoque une fonction de la librairie Chorus (la librairie des acteurs extérieurs au noyau), il exécute un appel à la fonction Trap() qui simule une trappe système vers le noyau Chorus (pour les acteurs Chorus, à travers des sockets).

Avant d'exécuter la trappe, Trap() vérifie qu'une connection existe bien avec Chorus. Si il n'en existe pas, elle considère que l'appelant est un acteur non initialisé et appelle donc actorInit() qui se charge d'établir une connection, de réserver la mémoire pour le contexte (c'est à dire augmente le segment data du processus de K_CTXTSIZE avec brk()).

3 Structure générale de COOL

Le système COOL est lui-même composé de plusieurs parties (figure 2) :

- un "noyau" contenant les structures de données locales à un site et les points d'entrée des primitives,

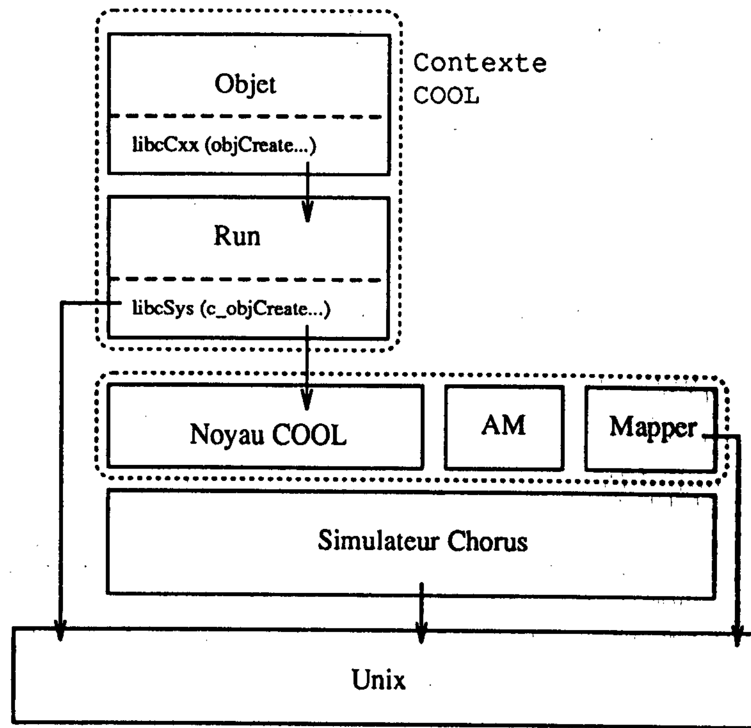

Figure 2 : Structure générale de COOL

- un Mapper chargé d'honorer les requêtes de page et le chargement des classes,
- un Availability Manager qui assure le suivi des messages pendant les migrations,
- les contextes utilisateurs lancés par le chargeur Run.

Le noyau Chorus et les principaux acteurs du système COOL sont chargés ensemble, de cette manière COOL partage l'espace du noyau ce qui lui permettrait sur un Chorus natif, de recevoir directement des trappes venant des acteurs utilisateurs. En fait, seul le "noyau" COOL a besoin de se trouver dans l'espace noyau, puisque le Mapper et l'AM sont des serveurs qui communiquent avec l'extérieur (en l'occurence avec COOL et le noyau Chorus) par messages.

Les contextes COOL communiquent avec le "noyau" COOL au travers des IPC Chorus simulés, par l'intermédiaire du noyau Chorus.

4 Les contextes COOL

Les contextes COOL (figure 3) sont plus ou moins équivalents aux acteurs Chorus à partir desquels ils sont construits. En respectant les abstractions Chorus, un contexte est une collection de régions de mémoire virtuelle et de threads. Dans le simulateur, cela correspond à des zones de mémoire et à des threads simulés.

4.1 La création des contextes COOL

La primitive contextCreate exécute un fork() puis un exec() du chargeur RUN avec pour argument le nom de la classe de l'objet à lancer (voir plus loin ce que fait RUN). Le Run fils récupère dans l'environnement le descripteur du socket vers son père et lui retourne par ce moyen la capacité de l'objet qu'il vient de créer.

Dans le simulateur Chorus, un contexte est un processus UNIX et comme tous les acteurs utilisateurs, il a une taille fixe (K_CTXTSIZE).

5 Les objets COOL

Les objets COOL (figuer 4) sont composés d'au moins un segment de code et d'un segment de données. Ces segments sont des régions de mémoire en principe virtuelle : dans le simulateur, il ne s'agit que de régions de mémoire dans l'espace d'un contexte COOL.

Les objets COOL peuvent être actifs (ou encore serveurs) : la propriètè d'objet serveur est caractérisée par la présence d'un appel à la primitive objReceive() dans le programme, celle d'objet actif par la présence d'une méthode main(). Les deux vont souvent de pair.

Cette partie de COOL est très liée au modèle d'objet de C++. Les objets sont désignés par un pointeur (la valeur de this) vers une structure contenant un pointeur vers le descripteur de l'objet et un autre vers la table des méthodes de l'objet.

5.1 La gestion mémoire des objets

Le segment de code d'un objet peut être partagé par plusieurs objets instances de la même classe. C'est une région de mémoire dont l'emplacement est déterminé à l'édition de liens (ld -T nn ...). C'est donc à l'utilisateur de bien choisir l'emplacement du code de chaque classe pour éviter les conflits.

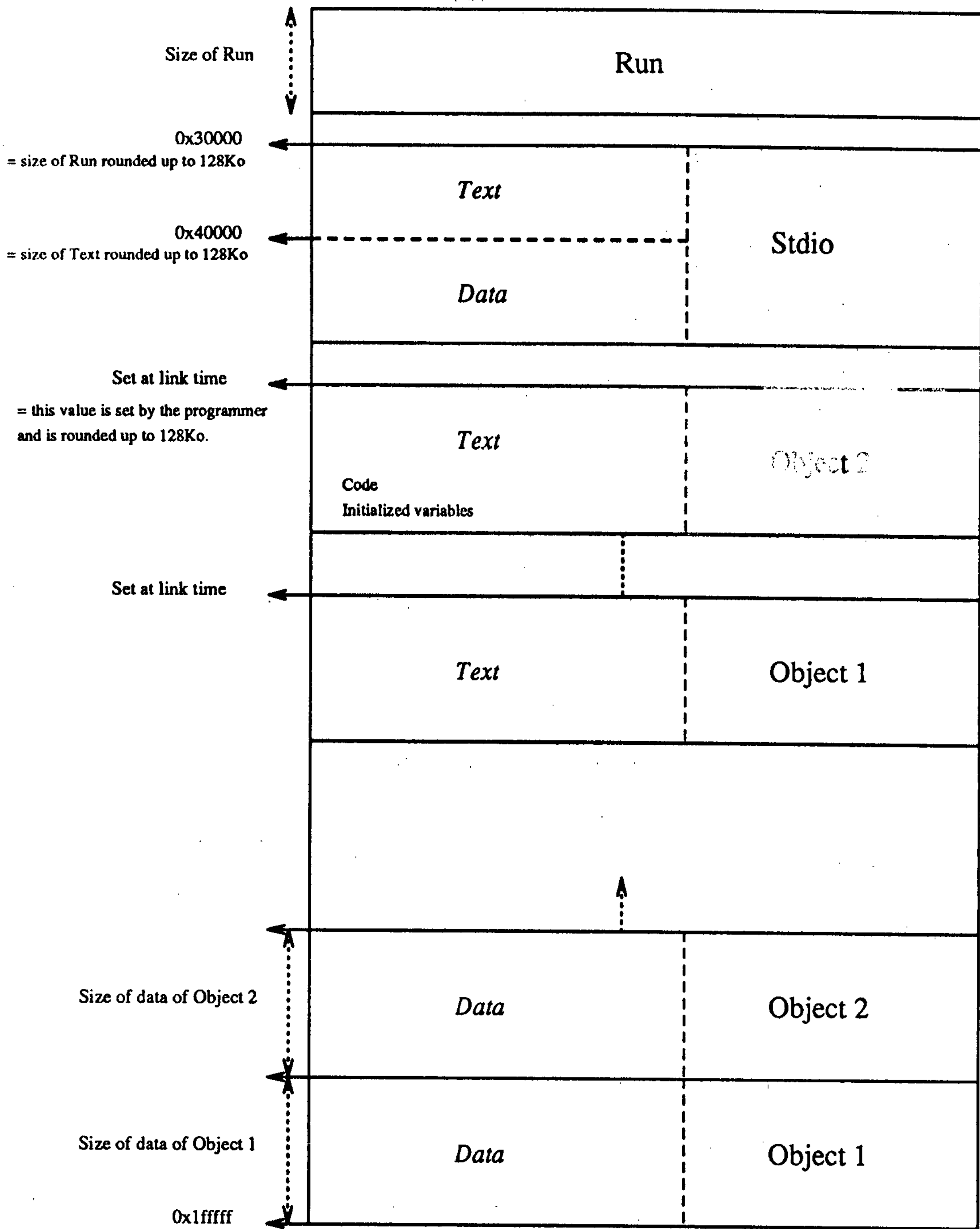

Figure 3 : Format d'un contexte

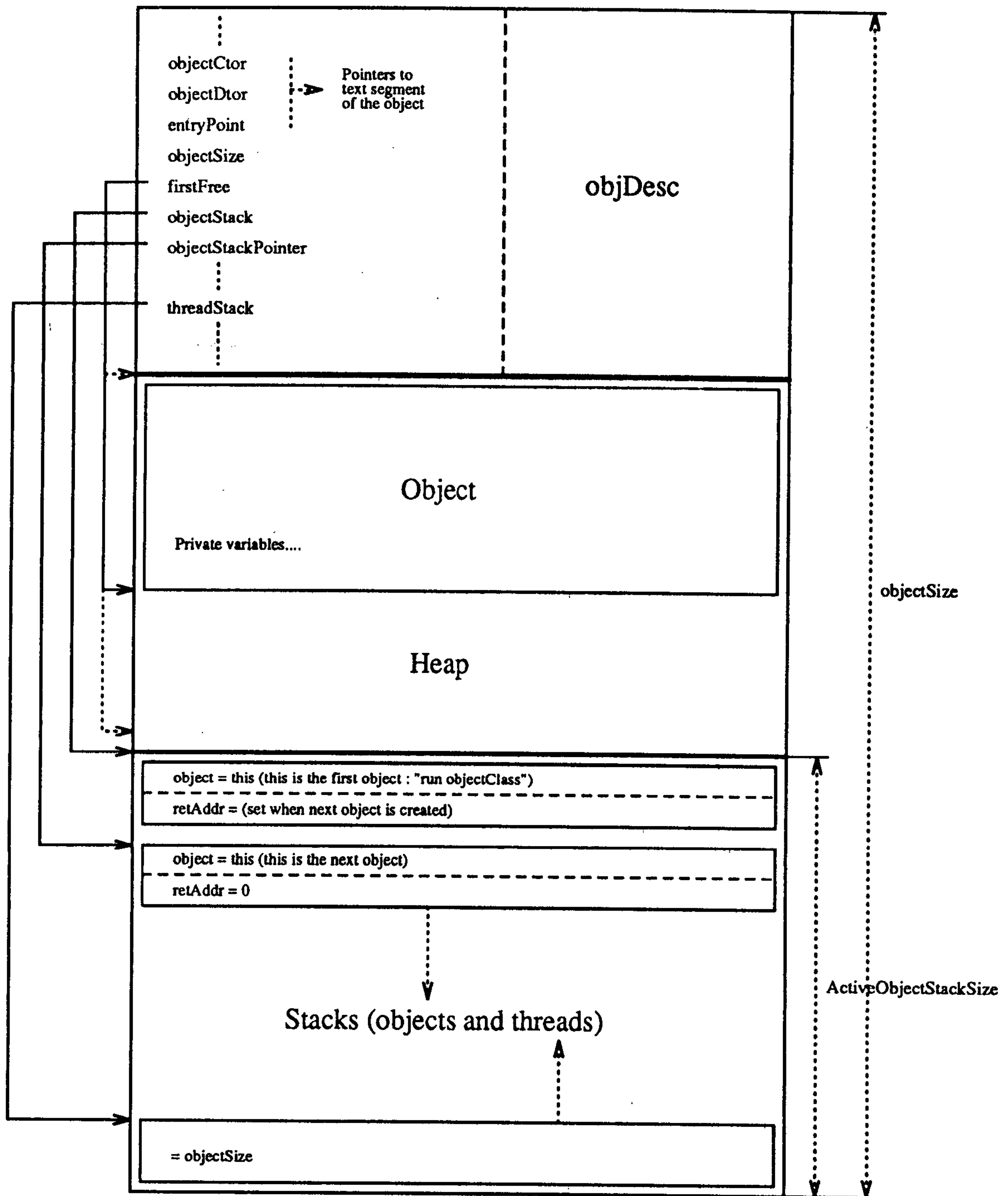

Figure 4 : Format d'un objet

Chaque instance possède un segment de données privé qui lui sert entre autres de tas pour l'allocation de mémoire. La taille du segment de données est fixée au moment de la création de classe (c'est à dire l'exécution de crClass sur la classe compilée). Elle peut être forcée à la valeur désirée, où prendre une valeur par défaut (DefaultObjectSize).

Les variables initialisées se trouvent dans le segment de texte. Il ne peut y avoir de variables globales non initialisées car COOL gère lui-même le segment data.

L'emplacement du segment de données est déterminé dynamiquement au chargement (au moyen d'un rgnMap() avec l'option K_ANYWHERE). L'adresse de ce segment peut changer au cours d'une migration dans un contexte différent.

Le segment de données est divisé en plusieurs parties :

- Un descripteur d'objet, contenant notament le bloc de données collecté par crClass et des informations concernant l'objet et son implantation en mémoire,
- Une zone libre de taille variable qui sert de tas.

Les données privées de l'objet sont allouées dans ce tas par son allocateur.

La librairie pour les objets COOL fournit un ensemble malloc-free qui gère cette zone de tas fixe en utilisant une stratégie best-fit à l'allocation et une stratégie de recollement à la libération.

Le segment de données contient également les piles des objets actifs.

5.2 Les piles des objets COOL

Dans le cas des objets actifs, le segment de données contient aux adresses les plus hautes :

- Une pile d'invocation qui enregistre le passage de l'exécution d'un objet à un autre,
- Une pile d'exécution propre à l'activité (thread) courante, qui contient à sa base un pointeur vers le descripteur d'objet.

La pile d'exécution est la pile standart du thread courant vers laquelle pointe le ponteur de pile (sp). A la base de cette pile se trouve un pointeur qui désigne le descripteur de l'objet actif dont dépend le thread courant. Un registre logiciel (selon la définition des contextes logiciels de Chorus) pointe également sur le descripteur d'objet d'une manière plus fiable.

La pile d'invocation d'un objet actif sert à retrouver l'objet courant : elle contient les adresses des descripteurs des objets traversés par le thread et les adresses de retour.

5.3 La création d'un objet COOL

La création d'un objet au sein d'un contexte COOL consiste en plusieurs opérations :

- La récupération du descripteur de classe (à travers le Mapper),
- L'initialisation des structures du noyau (création et insertion dans la table des descripteurs d'objet, insertion du descripteur d'objet dans la table des descripteurs de contextes),
- Installation de l'objet dans le contexte (mapping des segments etc...),
- L'initialisation des structures de données propres du contexte (principalement le descripteur d'objet),
- La création éventuelle d'une activité dans un état suspendu,
- Mise en place des piles d'invocation et du tas de malloc.

Ensuite on appelle le constructeur C++ de l'objet, qui alloue de la mémoire pour les données privées. Après l'exécution du constructeur, on échange la table de méthodes de l'objet avec une table générique pour intercepter les invocations locales, et on relance l'activité suspendue.

5.4 L'invocation locale entre objets

Ainsi, chaque invocation d'une méthode est convertie en un appel à une fonction (simpleInvoke() ou monitorInvoke() selon que l'objet doit s'exécuter en mode moniteur) avec pour argument l'index de la méthode dans la table originale. Cette fonction sauve l'adresse de retour et l'adresse du descripteur d'objet courant, puis appelle la méthode de l'objet, dont l'adresse est trouvée dans la table originale. Au retour elle pousse l'adresse de retour trouvée sur la pile d'invocation sur la pile d'exécution et revient. Ce mécanisme complexe est nécessaire pour avoir toujours l'adresse du descripteur d'objet courant à portée de la main. Cela rend nécessaire l'empilage de cette adresse à chaque invocation, et le dépilage au retour, d'où la nécessité d'intercepter les appels/retour par une fonction qui fasse les manipulations de pile nécessaires.

5.5 La migration d'objet

L'envoi d'un message peut s'accompagner de la migration d'un objet. La migration est faite en plusieurs étapes :

1. Dans le contexte de départ: empaquetage de l'objet (transforme les référence en relatif, applatissement de l'arbre),

2. Dans le noyau de départ: détachement ou copie des segments et des références locales, on met le descripteur d'objet dans le message, que l'on envoie au port de l'objet receveur,

3. Dans le noyau d'arrivée : on récupère le descripteur d'objet que l'on installe (mise à jour des structures de contexte...), on mappe les segments.

4. Dans le contexte receveur : on rétablit les références et les structures de données.

Lors de la migration d'un objet actif, l'exécution reprend au début de sa méthode main(). Le constructeur n'est pas rappelé. Il faut donc faire très attention lorsque l'on alloue des données dans le tas : elles ne sont jamais libérées alors que les références sont perdues. Pour éviter cela, il faut que les références soient conservées dans les variables privées de l'objet, et vérifier que l'on ne refait pas des allocations déjà faites (ou encore vérifier à l'aide d'une variable d'état si on est dans la première exécution de main()).

5.6 Les pointeurs relogeables

Lors de la migration d'un objet, l'adresse d'implantation de son segment de données peut varier. Pour cette raison, il est indispensable de déclarer des pointeurs relogeables que la librairie ajuste au moment de la migration pour conserver leur sens.

Les pointeurs relogeables sont réalisés à l'aide d'une classe qui redéfinit les opérations d'affectation et de référence, sa seule variable privée étant un index par rapport au début de l'objet courant (this).

5.7 Les entrées-sorties : l'objet stdio

A la création de chaque contexte, l'objet stdio est créé. Les entrées-sorties des objets COOL se font par invocation de l'objet stdio implanté dans leur contexte.

Cela permet de conserver les structures des données des fonctions entrées-sorties et donc l'état des entrées sorties.

Les fonctions de la librairie UNIX sont remplacées par des déroutements au travers d'une table qui les transforme en invocations locales des méthodes de l'objet stdio.

6 Le noyau COOL

Le noyau COOL réalise l'essentiel des primitives du système. La librairie de Run qui est son complément ne fait que s'occuper des mécanismes de membres, du relogement des pointeurs, des invocations locales. L'essentiel des fonctions réparties comme la migration ou les groupes d'objets, tout ce qui se rapporte à la mémoire virtuelle est fait par le noyau.

6.1 Structures de données

Le noyau COOL contient plusieurs structures de données, principalement:

- La liste des contextes,
- La liste des objets,
- La liste hashée des segments de texte.

Les descripteurs de contextes, d'objets et de segments de texte sont alloués dynamiquement dans des tableaux statiques situés dans le noyau COOL (encore appellés pools).

6.2 Segments

Les segments sont mappés par le noyau COOL au moment de la création des objets, ou au moment de leur installation après une migration.

Pour les segments de texte, le noyau vérifie qu'il n'est pas déjà dans la liste des segments mappés par l'un des contextes. S'il ne l'est pas, il est inséré dans la liste hashée, s'il l'est, le compte de références est incrémenté. Ceci permet de ne charger qu'une seule fois les segments de texte et de les mapper autant de fois que nécessaire dans des contextes différents ou à différents endroits.

7 Le Mapper COOL

Le mapper est un acteur indépendant qui communique avec le noyau comme avec le noyau COOL par l'intermédiaire de l'IPC Chorus (messages). Il remplit deux rôles distincts : celui de serveur de segments pour le noyau Chorus et celui de serveur de classes pour le noyau COOL.

Après avoir initialisé les deux parties, il lance un thread sur la boucle de service de segment et continue à exécuter la boucle de service de classes.

7.1 Le serveur de segments

Le noyau Chorus fait appel au serveur de segment dès qu'il a besoin de (mapper) une page d'un segment dans une région de mémoire. Sur une requête du noyau :

- création de segment : initialise les structures de données du noyau,
- page in : le serveur répond avec un message contenant la page du segment demandé,
- page out : le serveur écrit dans le fichier (segment) la page qui lui est transmise,
- destruction de segment.

7.2 Le serveur de classes

Le serveur de classe rèpond aux requêtes de descripteurs de classes provenant de la librairie de RUN. Au nom d'une classe, il répond en retournant un descripteur initialisé avec les valeurs des descripteurs de segments de texte et de données initiales. Pour chaque segment, on initialise sa capacité, et pour chaque région associée, on initialise l'adresse de début, ses options et sa taille.

8 L'Availability Manager

L'availability manager est un acteur Chorus chargé avec les autres acteurs système de COOL. Il est chargé de veiller à la bonne réception des messages au cours des migrations d'objets.

Lors de la migration d'un objet serveur COOL, il faut attacher le port de cet objet à un acteur afin qu'il puisse continuer à recevoir des messages. C'est ce que fait l'AM : au début de la migration, le noyau migre le port associé à l'objet vers l'AM local et lui envoie un message demandant de l'enregistrer. A la fin de la migration, lorsque l'objet est installé dans son contexte d'arrivée, le noyau du site destinataire envoie au premier AM une requête de récupération du port, l'AM fait alors migrer le port vers l'acteur destinataire.

Ce shéma pose cependant un problème : il requiert que Chorus fournisse un service de migration de ports inter-sites fiables offrant toute garantie quant au suivi et à l'ordre des messages, ce qui n'est pas le cas dans la version 3.3.

Il n'est donc pas possible d'assurer un service propre de migration de ports, et le rôle de l'AM a été court-circuité : on se repose sur les propriétés des groupes de ports Chorus.

9 Le chargeur COOL: RUN

Run est le chargeur des objets COOL : il met en place un contexte COOL et crée l'environnement d'exécution des objets actifs.

Il crée les structures de données associées au contexte dans le noyau en appellant c_contextCreate(). Il crée l'objet stdio (par objectCreate()), qui sert d'interface pour les entrées-sorties. Il crée l'objet de la classe donnée en argument (objectCreate()) et se charge d'initialiser la pile avec les arguments du constructeur.

C'est Run qui se charge du scheduling des threads dans les contextes. Quand il est lancé, il arme un timer (avec le signal SIGALRM) (valeur de 2 secondes). La fonction clock() appelée reçoit une structure contenant en particulier la valeur du pc et du sp. On appelle ensuite ThreadDelay() pour provoquer un rescheduling.

10 CrClass

Le rôle de crClass est de transformer les exécutables UNIX résultant de la compilation des classes C++ en objets utilisables par le chargeur RUN. Il lit la table des symboles et des chaines. Il y cherche un constructeur un destructeur et un main du nom de la classe (c'est à dire respectivement __*__ctor __*__dtor __*_main où "*" est le nom de la classe). Si il y trouve le symbole _objReceive, il suppose que l'objet est un objet serveur. Si il y trouve un symbole __*_main, il en conclut que l'objet est actif. Avec les informations qu'il a collecté, il construit une table qu'il ajoute à la fin de l'exécutable.

11 Un aperçu de COOL sur Chorus natif

Le portage de COOL sur Chorus natif permettra de clarifier de nombreux points en accentuant la distinction entre segments de texte et de donnés, en supprimant la plupart des limitations de mémoire que connaît le simulateur.

En particulier il sera possible de séparer les piles et les données en les plaçant dans des segments différents.

La structure de Run sera plus simple et plus naturelle, les threads étant apportés par Chorus et la mise en place des contextes étant largement basée sur la création des acteurs Chorus.

12 Conclusion

Cette documentation est nécessairement incomplète : sa réalisation a été faite dans un laps de temps assez court. Les éléments manquants ou peu clairs seront précisés ultérieurement, soit par une nouvelle version si cela est justifié, soit par des indications orales lors des séances de support.

Remerciements

La première version du prototype de COOL a été développée par Sabine Habert et Laurence Mosseri à l'INRIA et par Vadim Abrossimov chez Chorus Systèmes. Rodger Lea, également chez Chorus Systèmes, a largement contribué à la mise au point en écrivant les premières applications sur COOL. Marc Shapiro et Marc Guillemont ont donné leur temps et leur expérience lors des discussions sur le modèle de COOL. L'équipe du projet CIDRE au SEPT est responsable de la conception de l'environnement C++.

A Constantes du système COOL

A.1 Partie noyau

sys/src/c_ipc.cxx :

- C_MAXMSGSIZE 8192 Taille maximale d'un message (suivant celle de Chorus)
- MAXMSGNB 64 Nombre de messages utilisables à chaque instant (dimension des tableaux et des Pools)

sys/src/sCool.h :

- MAXOBJSITE 512 Nombre de descripteur d'objet par site (et donc d'objet) (dimension des tableaux et pools).

sys/src/Tseg.h :

- tsegMaxNb 16 Nombre des descripteurs de segments de texte disponibles
- MAXCVMDESCS 64
- MAXCTXTS 8 Nombre de descripteurs de contextes disponibles (dimension des tableaux et pools)

sys/include/cool.h :

- ObjectDataMaxSize 0x1000 Inutilisé
- PACKDESC_SIZE 236 Taille d'un descripteur d'objet compressé
- UMSGMAXSIZE MAXMSGSIZE 8192 Taille maximale théorique d'un message utilisateur (la taille maximale réelle dépend du nombre d'objets transmis)
- MAXOBJS 128 Nombre maximal d'objets par contexte
- MAXGROUPS 8 Nombre maximal de groupes dont un objet peut être membre

A.2 Partie librairie / chargeur

cxx/src/context.hxx :

- MAXMEMBERS 16 Nombre maximal de membres qu'un objet peut avoir (à un niveau)

cxx/src/crParms.h :

- DefaultObjectSize 0x1000 Taille du segment de données d'un objet par défaut, non comprise la pile
- ActiveObjectStackSize 0x2000 Taille de la pile par défaut

cxx/src/AM.cxx :

- MAXAMentries 64 Nombre maximal d'objets en transit (pas utilisé)

B Liste des codes d'erreurs de COOL

C_ENORES	-14	No more resource, epuisement de memoire, de table, signale aussi des erreurs de ressources UNIX
C_ENOCAP	-15	No capability
C_EBRDOBJ	-16	Broadcast d'un objet
C_ENOTIMP	-17	Not implemented, pas realise
C_EPERS	-18	Objet persistant
C_EBADOBJ	-19	Verrouillage impossible
C_EBADATTACH	-20	Tous les objets n'ont pas etes bien installes
C_EUNIX	-21	Mauvaise entree dans le service de nom
C_EMBR	-22	On tente d'appliquer une operation interdite sur un objet membre
C_ESELF	-23	(cxx/src/ipc.cxx)
C_ECURR	-24	Jamais utilisee
C_EBADFMT	-25	Classe pas en format a.out
C_EINVAL	-26	Invalid value supplied, valeur hors du domaine de definition,
C_EFAULT	-27	svCopy failed
C_EMSGTOOBIG	-28	Message utilisateur trop grand
C_EMAXGROUP	-29	Objet membre de trop de groupes a la fois

Table des matières

1 Introduction **1**

2 Le Simulateur Chorus **1**

2.1 Les librairies Chorus 1

2.2 Création d'un acteur Chorus sur le simulateur UNIX 2

3 Structure générale de COOL **2**

4 Les contextes COOL **4**

4.1 La création des contextes COOL 4

5 Les objets COOL **4**

5.1 La gestion mémoire des objets 4

5.2 Les piles des objets COOL 7

5.3 La création d'un objet COOL 8

5.4 L'invocation locale entre objets 8

5.5 La migration d'objet 8

5.6 Les pointeurs relogeables 9

5.7 Les entrées-sorties : l'objet stdio 9

6 Le noyau COOL **10**

6.1 Structures de données 10

6.2 Segments 10

7 Le Mapper COOL **10**

7.1 Le serveur de segments 11

7.2 Le serveur de classes 11

8 L'Availability Manager **11**

9 Le chargeur COOL: RUN **12**

10 CrClass **12**

11 Un aperçu de COOL sur Chorus natif **12**

12 Conclusion **13**

A Constantes du système COOL **14**

A.1 Partie noyau 14

A.2 Partie librairie / chargeur 14

B Liste des codes d'erreurs de COOL **16**

Références

[Abrossimov 1989] Abrossimov (Vadim), Rozier (Marc) et Gien (Michel). – *Virtual Memory Management in Chorus.* – Rapport technique n° CS/TR-89-30.1, Saint-Quentin-en-Yvelines (France), Chorus-Systèmes, mai 1989.

[Armand 1989] Armand (François), Herrmann (Fréderic), Lipkis (Jim) et Rozier (Marc). – *Multi-Threaded Processes in Chorus/MIX.* – Rapport technique n° CS/TR-89-37.3, Saint-Quentin-en-Yvelines (France), Chorus-Systèmes, octobre 1989.

[Deguin 1989] Deguin (Anne), Bourdon (François), Deshayes (Jean-Marc), Greard (Marcel), Tourrade (Didier) et Touzeau (Pierre). – Cidre: Intelligent circulation of distributed folders. *In : International Workshop on Telematics.* – Denver (CO), (USA), 1989.

[Habert 1989] Habert (Sabine). – *Gestion d'objets et migration dans les systèmes répartis.* – Paris (France), Thèse de doctorat, Université Paris-6 Pierre-et-Marie-Curie, décembre 1989.

[Habert 1990] Habert (Sabine), Mosseri (Laurence) et Abrossimov (Vadim). – *COOL: Kernel Support for Object-Oriented Environments.* – Rapport technique n° 1211, Rocquencourt (France), INRIA, avril 1990.

[Rozier 1988a] Rozier (M.), Abrossimov (V.), Armand (F.), Boule (I.), Gien (M.), Guillemont (M.), Herrmann (F.), Kaiser (C.), Langlois (S.), Léonard (P.) et Neuhauser (W.). – Chorus distributed operating systems. *Computing Systems*, vol. 1, n° 4, 1988, pp. 305–367.

[Rozier 1988b] Rozier (M.), Abrossimov (V.), Armand (F.), Gien (M.), Guillemont (M.), Hermann (F.), Kaiser (C.), Leonard (P.), Langlois (S.) et Neuhauser (W.). – *Overview of the Chorus Distributed Operating System.* – Rapport technique n° CS/TR-88-7, Montigny-le-Bretonneux (France), Chorus Systèmes, juin 1988.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique