"Architektur von Rechensystemen", 12. GI/ITG Fachtagung, Kiel, 23.-25. März 1992
A. Jammel (Ed.), Springer Verlag, Berlin Heidelberg New York

Clans & Chiefs

Jochen Liedtke
German National Research Center for Computer Science (GMD)
W – 5205 Sankt Augustin
Germany

jochen.liedtke@kmx.gmd.dbp.de

Abstract:
Clans are introduced as a basic concept of an operating system kernel. They permit full algorithmic control of process interaction in a user definable but secure way. All communication across a clan's borderline is inspected and possibly modified by the clan's so called *chief* task. Thus the mechanism can be used for protection, remote communication, debugging, event tracing, emulation, connecting heterogeneous systems and even process migration. It has been implemented in the operating system L3 where, besides some smaller applications, remote message handling and a multi-level-security monitor rely on clans.

Keywords: Clan, chief, inter-process communication, operating system, protection, L3.

1. Introduction

Offering protection mechanisms is one of the most important tasks of an operating system. Ideally the mechanisms should be very few but powerful, permit arbitrary strong or weak protection policies and restrict the users as little as possible. In modern μ-kernel based system architectures various security policies and high level mechanisms are built on top of the kernel as servers: authentication servers, file servers restricting access by reference monitors or access control lists and others. It is the μ-kernel's job to support the servers by basic protection mechanisms, usually *autonomy* (isolation of tasks/address spaces) and some sort of *process interaction/communication control*. Examples of such existing client/server based systems are Amoeba [15] (tickets/capabilities), Mach [1,14] (controlled channels/port rights) or BirliX [6,7] (message integrity). All these examples are "message-oriented". This model seems to be more convenient in the context of multiprocessors and distribution than the "procedure-oriented" one. Lauer and Needham pointed out "that these two categories are duals of each other and that a system which is constructed according to one model has a direct counterpart in the other" [8]. However, the message-oriented model will be used for reasoning here.

The idea of Clans & Chiefs was influenced by the subject restriction concept of BirliX [7]. Here it is possible to enwall suspicious active entities (subjects) by means of subject restriction lists. Each of these specifies the set of all partners (mostly servers), which are accessible by the corresponding suspicious subject. The clan concept goes beyond this in two aspects:

- It allows full algorithmic control on user level. Arbitrary algorithms can supervise and modify the interactions of active entities.
- Clans form hierarchies of protection as well as of semantic domains.

Reasoning and implementation are based on the operating system L3 [10], which is purely message oriented (like RC4000 [4], Demos [2], Mach [1] and many others). In L3 autonomous tasks with protected address spaces communicate exclusively via messages.[1] In contrast to Demos and Mach messages are transferred directly without intermediate system objects like links or ports (for reasons see section 2).

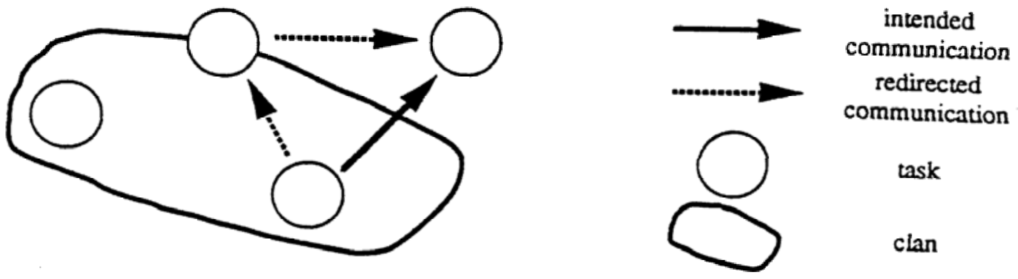

Figure 1. Clan & Chief

In this context a *clan* is defined to be a set of tasks headed by a *chief* task. All messages from clan members to tasks outside the clan are redirected by the kernel to the chief (see figure 1). The same happens to messages coming from outside. Thus the chief controls all communication of the clan with the outer world.
It turned out that the clan concept is not only useful in the area of protection. Besides debugging and tracing, chiefs can be used to forward messages via networks, to en/decrypt messages via suspicious channels, to emulate environments or hide heterogeneity by message transformation. Furthermore they cannot only protect the outer world against suspicious subjects but also their clans against the outer world, independent of application programs.

A simple example using the clan mechanism is the prototype of a virus encapsulator. It uses the following strategy: Each time suspicious software is invoked, it creates a clan with the virus encapsulator as its chief. Inside the clan the suspcious program runs like in a cage:

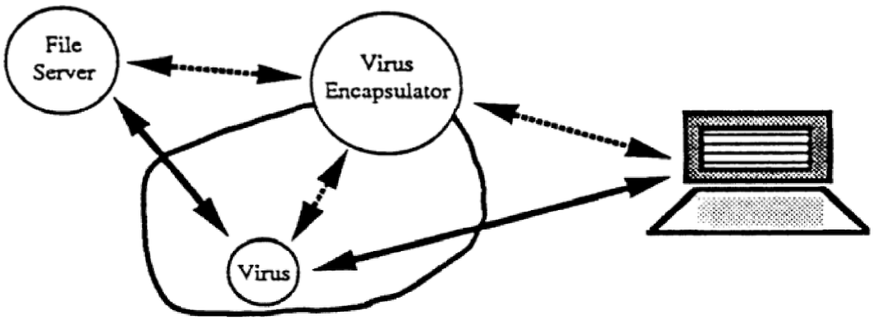

Figure 2. Encapsuled Virus

Here the chief algorithm is not very complicated:

```
virus encapsulator (program, read file list, write file list, terminal) :
  generate and start (clan member, program) ;
  REP
      wait for (message, sender, receiver) ;
      IF incoming message
      OR message to terminal
      OR read access to read permitted file
```

[1] Similar to Mach shared objects are realised by external pagers. Thus [18] access to memory objects is also based upon communication.

```
        OR write access to write permitted file
            THEN send deceiving (message, sender, receiver)
            ELSE report illegal message and drop
        FI
    ENDREP .
```

Besides this and some further small examples (ipc tracer, driver debugger, version adapter) two larger systems applying Clans & Chiefs have been developed on top of the L3 μ-kernel. The first handles remote inter-process-communication ("ipc") by means of chiefs as shown in figure 3:

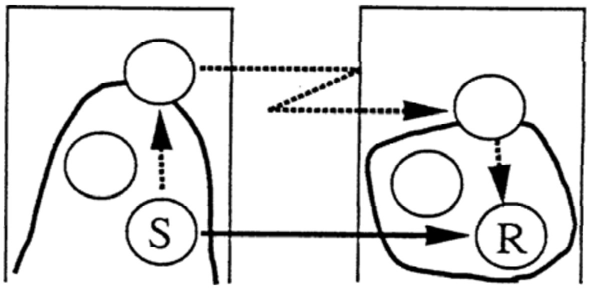

Figure 3. Networks and Clans

The second system implements multi-level security (B3/A1 functionality in [16] or F5 in [19]). Both profit from the fast ipc of L3: 25.0 μs (386, 25 MHz) or 12.3 μs (486, 25 MHz) for short messages. The performance is discussed in more detail in [12].

Section 2 briefly describes L3, which is the basic model used for reasoning. (L3 does exist and is used commercially.) Section 3 defines the concept of clans in detail and discusses security problems related to it. Section 4 describes the current implementation of the clan concept in L3. Section 5 discusses the possibilities implementing the clan concept in operating systems like Demos or Mach.

2. The μ-Kernel Concepts Used for Reasoning (L3)

Nowadays many more or less different kernel philosophies (and terminologies) are in use. To avoid confusion and misunderstanding the argumentational basis used throughout this paper is described briefly. In fact, this is the μ-kernel of an existing and commercially used operating system called L3 [3,9,10].

The L3-kernel is an abstract machine implementing the data type *task*. A task consists of

- at least one *thread*
  A thread (like a Mach thread [1]) is a running program. All threads (actually up to 16384 per station), except resident (unpaged) driver or kernel threads, are persistent objects.
- up to 16383 *dataspaces*
  A dataspace is a virtual memory object of size up to 1 GB. Dataspaces are also persistent objects and are subject to demand paging. Copying and sending is done lazily. Physical copy operations are delayed ("copy on write") like in Accent [17], Mach [1], EUMEL [5] and BirliX [6].
- one address space
  Dataspaces are mapped dynamically into the address space for reading or modifying. For hardware driver tasks the address space is logically extended by the IO ports assigned to the task. (All device drivers are located outside the kernel and run at user level.)

As in Mach paging is done by the default or external pager tasks. So all interactions between tasks and with the outer world are based on inter-process-communication.

The original ipc concept of L3 is quite simple. Active components, i.e. the threads, communicate via messages consisting of strings and/or dataspaces. Each message is sent directly from the sending to the receiving thread. There are neither communication channels nor links, only global thread and task identifiers (ids).[2)] Thus opening is not necessary for ipc. Furthermore no additional kernel objects such as links or ports are involved. This simple model differs in two points from most other message oriented systems:

- absence of explicit binding (opening a channel)
  One could expect additional costs, because the kernel must check the communication's validity each time a message is sent. But on the other hand there is no necessity for the bookkeeping of open channels. We consider the simple concept as a bit more elegant and, in fact, communication in L3 is faster than usual [12]. Nevertheless some special types of clans may be realised more efficiently when using explicit binding (see section 5).
- no message buffering
  Due to the absence of channel objects we have synchronous ipc only; sender and receiver must have a rendezvous. But practice has shown that higher level communication objects like pipes, ports, queues and others can be implemented flexibly and efficiently on top of the kernel by means of threads.

For local[3)] ipc the kernel guarantees *message integrity*:

• *no deceit:*	A receiver will always get the correct sender id.
• *no wiretapping:*	Only the receiver specified by the sender sees the message.
• *no modification:*	Messages will be neither modified nor lost.

Tasks and threads have unique identifiers, unique even in time. Thus a server usually concludes from the sender id of the order message whether the required action is permitted for this client or not.

So the *message integrity* in connection with the *task autonomy* is the basis for higher level protection, always implemented on top of the μ-kernel.

3. Clans & Chiefs

Within the L3 scenario and also other systems based on direct message transfer, e.g. BirliX, protection is basically a matter of message control. For the well known access control lists (acl) this can be done at the server level. But maintenance of large distributed acls becomes hard, when access rights change rapidly. So Kowalski and Härtig [7] propose to complement object (= passive entity) protection by subject (= active entity) restrictions. In this concept the kernel is able to restrict the outgoing message of a task (the subject) by means of a list of permitted receivers.

The clan concept is an algorithmic generalization of this idea, or of reference monitors respectively.

[2)] How to get the id of a new partner for communication? This is not a job of the kernel but of some name server(s). Usually the creating task implants the id of at least one name server into the new task. By this it can communicate with at least this name server to get further task/thread ids.

[3)] The old remote ipc of L3 is no longer relevant here, because first it is not needed for the following argumentation and second it is replaced by an application of the new clan concept described in this paper.

A *clan* is a set of tasks headed by a *chief* task. Inside the clan all messages are transferred freely and the kernel guarantees message integrity. But whenever a message tries to cross a clan's borderline, regardless whether outgoing or incoming, it is redirected to the clan's chief. This chief may inspect the message (sender and receiver as well as contents) and decide whether or not it should be passed to the destination to which it was addressed. As demonstrated in figure 4 these rules apply to nested clans as well.

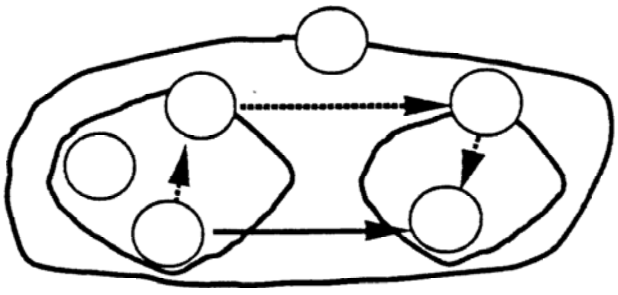

Figure 4. Nested Clans

Obviously subject restrictions and local reference monitors can be implemented outside the kernel by means of clans. Since chiefs are tasks at user level, the clan concept allows more sophisticated and user definable checks as well as active control.

3.1 Creation

The clan concept leads to a recursive data type *task*. Important new operations are

. *create inner task*
. *delete inner task*

If a task creates its first inner task, a new clan with the invoker as its chief is generated. Then the newly created task and the chief are the members of the clan. By further creating inner tasks the chief will extend the clan. Thus chiefs are always members of two clans, the *inner clan* headed by the chief and the *outer clan* containing it as a simple member.

If *delete inner task* is applied to a chief of a clan, all clan members will be deleted implicitly. Deletion could be handled more freely, but this strict method prevents clan members from unrecognized changes of their chief (see 3.5).

It is important to understand that *create inner task* differs significantly from the conventional *create child task* operation. The task tree does not necessarily reflect the clan structure. Creating a child task generates a new member of the invoker's clan and does not generate a new clan. Thus the tree of tasks generated by creation of child tasks is not necessarily kernel based like clan/chief. Usually the task tree is a higher level object. The most natural way for implementation is by using a clan's chief as server for child task creation and deletion. If a task A wants to create a new child task, it sends an appropriate message to *chief (A)*; this one creates a new clan member B by invoking *create inner task* and updates the task tree by entering B as child task of A.

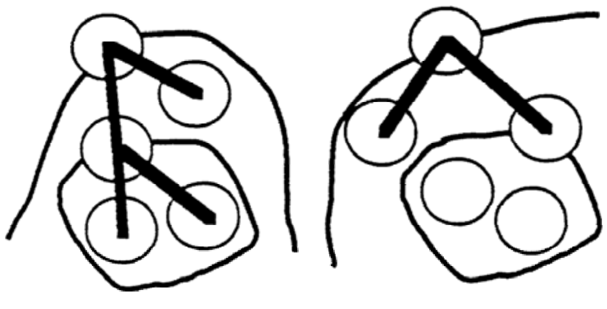

Figure 5. Task Tree and Clan

So task trees are clan dependent. A chief can isolate its inner clan from the outer clan and hide its task tree and name space (figure 5, right). Also complete integration of the inner clan members into the outer clan's task tree and name space is possible by cooperation of the chiefs (figure 5, left).

3.2 Communication

It has now to be considered, whether chiefs can be implemented as tasks or threads, or whether even a new type of object is needed.

Chiefs in general need to protect their autonomy against the objects they supervise. Therefore the given concept hierarchy of thread and task is extended towards clans: clans contain tasks, tasks contain threads. As a consequence chiefs are introduced as tasks, not as threads. To enable redirection of messages to chiefs, ipc to a task is introduced meaning sending to an arbitrary thread of the task:

> Let $\mapsto$ denote a message transfer. Given a thread T and a task A, $T \mapsto A$ is defined as $T \mapsto T_A$, where T_A is an arbitrary thread of A.

This allows simple implementations of the clan concept, which always redirect messages to a dedica thread of the chief. As well, more sophisticated implementations are possible by chosing dynamically the receiving thread of the chief. Correspondingly the chief of a thread T_A being part of task A is defined to be chief (A).

These agreements make it convenient to use variables like S, R, C in the following both for tasks as well as for threads.

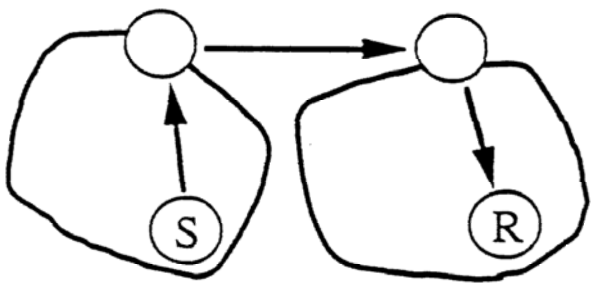

Figure 6. Inter Clan Message Transfer

Messages crossing clan borderlines are redirected to the chief controlling the clan. So sending a message from S to R (figure 6) is split into three steps:

(1)	S	$\mapsto$	R	*redirected to* chief (S)
(2)	chief (S)	$\mapsto$	R	*redirected to* chief (R)
(3)	chief (R)	$\mapsto$	R	

To allow such communication, thread ids must be visible outside the clan, where the thread resides. So clans form a hierarchy, but thread and task ids form a flat space as far as visibility is concerned.

For a more formal specification some relations are defined:

> $A \approx B$, *A and B belong to the same clan,* iff[4)]
>
> $\mathrm{chief}(A) = \mathrm{chief}(B) \vee \mathrm{chief}(A) = B \vee A = \mathrm{chief}(B)$

[4)] Unfortunately $\approx$ is no equivalence relation, because chiefs necessarily belong to their inner and outer clans.

$A \prec B$, *A lies inside B's inner world,* iff[5)]

$\text{chief}(A) = B \vee \text{chief}(A) \prec B$

The relation $\not\prec$ (*outside* the inner world) is the exact negation of $\prec$. (Thus $A \not\prec A$ holds.)

The kernel's redirection policy applied to an ipc request $S \rightarrowtail R$ can be formally described by:

$S \rightarrowtail \text{chief}(S)$	*if* $R \not\prec \text{chief}(S)$
$S \rightarrowtail R$	*if* $S \approx R$
$S \rightarrowtail C$	*if* $R \prec C \approx S,\ C \neq \text{chief}(S)$

Due to the strict hierarchy of clans the chief C is unique.

3.3 The Restrictive Model

In the so called restrictive model chiefs are only allowed to

- inspect a redirected message,
- throw it away,
- forward it unchanged to the true destination, which is always informed about the true sender.

Furthermore the chief cannot store messages. (This restriction seems to be unnecessary, if all application level protocols are resistent to replays.) No deceit is possible by chiefs, only suppressing communication and listening.

Thus the kernel guarantees a certain level of security, even when chiefs are involved in ipc. But due to the possibility of secret wiretapping this advantage seems to be rather small. On the other hand the application of clans and chiefs in this model is restricted to protection, consistency checking and tracing. Other possibilities require more liberal features. Moreover, the implementation of messages as protected kernel objects would require a substantial overhead. So this restrictive model is dropped.

3.4 The Liberal Model

The liberal model allows chiefs to deceive in a controlled way. When sending into its clan, a chief may specify itself or any thread of the outer world as the apparent sender. When sending to the outer world, it may specify itself or any thread of its inner world as the apparent sender.

The notation $C|_S \rightarrowtail R$ is used to denote that C sends a message to R simulating S to be the sender. Given S, R and a chief C, *direction preserving deceit* happens if either

$C|_S \rightarrowtail R \quad S \not\prec C,\ R \prec C$ (incoming)

or

$C|_S \rightarrowtail R \quad S \prec C,\ R \not\prec C$ (outgoing)

The liberal model allows chiefs to deceive, but only direction preserving. $C|_S \rightarrowtail R$ with $S,R \prec C$ or $S,R \not\prec C$ $(S \neq C)$ are not permitted. The latter ones would be undetectable (!) "interventions by a third party" into the inner or outer world, thus corrupting all security policies.

Furthermore chiefs are free to modify or drop messages. So it is no longer necessary to implement

5) $\prec$ is a partial ordering.

messages as protected kernel objects. They can be regarded as simple values. The consequences concerning integrity are discussed in the next section.

Now chiefs are real algorithmic filters usable for checking as well as for adapting and transforming incoming and outgoing messages. They permit using clans more or less transparently concerning the ipc of the other tasks and threads. It is hoped that this is a good basis for building higher level abstractions on top of the clan concept.

3.5 Trustworthiness and Integrity

Within a clan the integrity of messages is guaranteed by the kernel. But as soon as borderlines are crossed, potentially suspicious chiefs are involved. So the sequence of chiefs involved in a particular message transfer determines its integrity.

> The sequence of the sender S, the chiefs involved in transfer and the receiver R is called the *communication path* $S \gg R$.

A communication path is determined uniquely by S and R, assuming inter-clan migration of tasks not supported. The trustworthiness of a communication path is given by the trustworthiness of the chiefs. To simplify argumentation two qualities are defined:

> A chief is called *P-trusted*, if it is trustworthy with respect to some given predicate P.
> It is called *P*-trusted*, if it is P-trusted and communicates for P-related operations only with other P*-trusted chiefs or within its own clan.[6)]

Sometimes the use of P*-trusted chiefs can solve the integrity problems in a very elegant way. But chiefs cannot be P*-trusted for all predicates P and suspicious chiefs are needed, e.g. to represent insecure data channels or less trusted domains. Thus a general method is required which allows sender and receiver to decide about the trustworthiness of the communication path.

For each real transfer $S \leftarrowtail R$ there are the originally requested transfer $S \leftarrowtail R'$, where R′ is the receiver specified by S, and the transfer $S' \leftarrowtail R$ as it arrives at the real receiver considering S′ as sender. At a first glance it seems to be necessary for R to determine $S' \gg R$ and for S to determine $S \gg R'$ in a secure way. These are symmetric problems; so we will discuss the receiver's part judging about $S' \gg R$.

Obviously an incorruptable kernel function *chief (A)* would do the job. Unfortunately this wou' require the network handling for the remote case to be part of the μ-kernel. But just the network is a typical application of the clan concept (figure 3).

Instead of the general chief function the kernel is able to provide two simple functions, *rchief*, which is restricted to the function invoker's clan, and *nchief* yielding the chief nearest to the function invoker in the direction of the addressed task. Suppose R has received a message from S. When R invokes the kernel functions, it gets

$$\text{rchief}_R\ (S) = \begin{cases} \text{chief (S)} & \textit{if } S \approx R \\ \text{fails} & \textit{otherwise} \end{cases}$$

$$\text{nchief}_R\ (S) = \begin{cases} \text{rchief (R)} & \textit{if } S \not\prec R \\ C \neq \text{chief(S)} & \textit{if } S \prec C \approx R \end{cases}$$

6) At a first glance the restriction to other P*-trusted chiefs seems to be stronger than restricting the complete communication paths to P-trusted chiefs. But in fact this can be adjusted by modifying P.

Due to the clan hierarchy C is unique. *nchief* is counterpart to the function the kernel uses for redirection.

By means of this R can try to parse $S' \gg R$ backwards:

```
determine path (S,R) :
  IF in same clan (S,R)
    THEN S
  ELIF L*-trusted (nchief (S))
    THEN send ("chief(S)?", nchief (S));
         receive (path of chiefs) ;
         nchief (S) CAT path of chiefs
    ELSE nchief (S)
  FI .

L : "uses the above algorithm" .
```

A chief trustworthy with respect to the predicate L is said to be *location trusted*. The algorithm yields the *relevant postfix* of $S' \gg R$ leading back either to the first not location trusted chief or to S, if there is no location suspicious one in $S \gg R$. It is proposed that the receiver classifies all paths, which are not completely location trusted, as suspicious with respect to all other predicate P too. In practice this seems to be no hard restriction. So specific integrity decisions can be done based on the relevant postfix. Some interesting properties of this integrity judgement are:

- Usually one receiver or one sender has to judge the message transfer integrity only once. If the path from S to R is of integrity, no new chiefs can intrude into this subsystem. This feature is weakened when introducing migration.
- In many cases a very elegant way for solving the integrity judgement problem may be to use a chief. If the chief drops all messages coming via suspicious paths you have a fairly general solution outside the applications.
- Of course sometimes suspicious paths have to be used. They can be made secure by use of data encryption techniques. If this is done in trusted chiefs, the domain of P-trustworthiness can be widened.

3.6 Inter-Clan Migration

The model described so far is very static. Once generated a task is fixed in the clan hierarchy. This feature is very convenient when reasoning about security and integrity. But if tasks were permitted to migrate to a different clan, process migration (between computers) could be implemented by means of clans and chiefs.

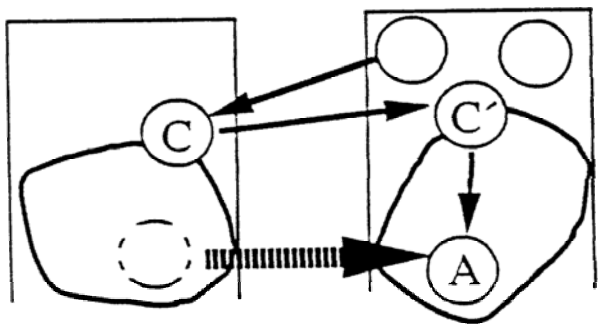

Figure 7. Migration

In figure 7 a task A has migrated to a different station. The messages are redirected automatically by the clan system. The new chief C′ cooperates with the old chief C and emulates the original environment for A.

When the migration feature is added to the model, the analysis of communication paths described in 3.5 becomes uncertain. Between or even during message transfer a task may migrate. So previous or subsequent analysis of $S \gg R$ may give invalid results.

These difficulties can be overcome by adding *migration numbers* to task and thread ids, which are incremented at each inter-clan migration. As a consequence the receiver of a message will realise that the source has moved in the meantime and can analyse the new communication path. The problems of finding a task jumping from clan to clan and judging integrity in such a situation are subjects of current research (see [13] for more details).

4. Implementation in L3

Clans & Chiefs were implemented in L3 to prove that the concept can be realised efficiently and provide experience for further experiments concerning the usefulness, convenience and applicability of the clan concept. This section briefly describes the extent of the implementation and the experience obtained to date.

The clan concept described in section 3 except the migration features was implemented. Migration was dropped, because up to now L3's thread ids are not location independent. We expected significantly higher implementation costs and performance problems. So we decided first to do without migration.

As mentioned in section 2, the original L3 ipc is quite simple. Its primitives are

. send (⟩dest, ⟩message, ⟩send timeout, res⟩)[7)]
. receive (⟩source, message⟩, ⟩rec timeout, res⟩)
. wait (message⟩, source⟩, ⟩rec timeout, res⟩)
. call (⟩dest, ⟩message⟩, ⟩send timeout, ⟩rec timeout, res⟩)

Receive accepts only messages from the source specified and *Wait* accepts messages from any source. *Call* works like

```
send (dest,message,sendtimeout,res) ;
IF res = ok
  THEN receive (dest,message,rectimeout,res)
FI
```

but is atomic, in contrast to the above sequence of system calls. This allows a simple Remote Procedure Call (RPC) implementation.

Obviously an extension to the μ-kernel level is necessary to implement clans and chiefs. Besides creation and deletion of inner tasks one new ipc primitive was added to the kernel:

. send (⟩dest, ⟩apparent sender, ⟩message, ⟩send timeout, res⟩)

Here the kernel accepts direction preserving deceit only. Furthermore the kernel provides

. rchief (⟩task, chief⟩, success⟩)

. nchief (⟩task, chief⟩)

[7)] ⟩x denotes an input, x⟩ an output and ⟩x⟩ an inout parameter.

Without change of the kernel interface the ipc semantics were extended as described in section 3, i.e. messages crossing clan borders are redirected to the corresponding chief.

This implementation was done during a redesign phase of the μ-kernel. So the manpower needed to implement just the clan concept is hard to isolate, but certainly less than 3 person weeks were needed for pure implementation.

5. Clans in other Message Oriented Operating Systems

Due to its structuring effect and its features related to integrity and trustworthiness, the clan mechanism goes beyond the redirection of messages already available in Demos [2]. But systems like Demos or Mach [14], which permit redirection by kernel level channels (Demos: "link", Mach: "port right") *and* control of the distribution of these channels, should allow the implementation of clans on top of them. This will be outlined using the Mach terminology.

The principle idea is to use alias ports in the chief task for all ports belonging to the outer world. Then the chief controls all outgoing and incoming communication:

a) Whenever a chief task creates a new clan member, the initial port rights of the new task all have to identify ports of the chief. Then requesting new ports will be done via the chief.

b) Whenever a port right is sent to a clan member from outside or sent outside from a clan member, the chief generates an alias port and sends rights on this instead.

c) Whenever a port right is sent from one clan member to another, the chief may forward it. Thus intra-clan communication will not use the chief.

The chief must not pass port rights between its inner and outer clan directly, only by means of aliases. If there is only one connection from the clan to the outer world, which is not controlled by the chief, arbitrary port rights, i.e. new uncontrolled connections, can intrude into the clan. Thus this model also enforces a hierarchical structure of clans. It is not yet clear, whether the judging of integrity in such an implementation can be done similarly to in the original clan implementation.

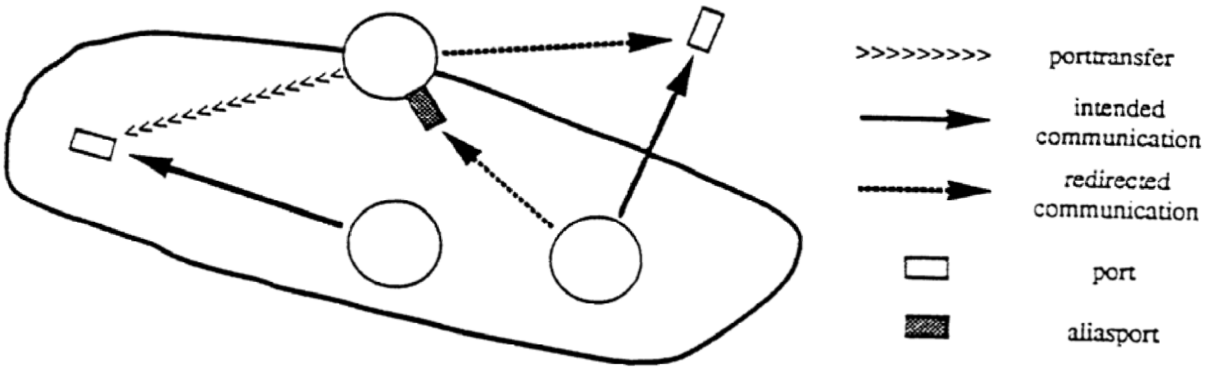

Figure 8. Clans in Mach

A picture (figure 8) of this model looks very similar to figure 1. Perhaps performance benefits can be expected for some types of clans: If a chief only decides on the basis of the sender-receiver pair about the communication's legality, the chief is only involved at binding time, not at communication time. But whenever messages have to be inspected or modified or communication rights change dynamically, this bind time bonus disappears.

6. Conclusion

Some conclusions can be drawn but should be regarded as preliminary, because increasing experience of Clans & Chiefs will give new arguments or change the importance of old ones.

- The clan concept can be implemented efficiently.
- Clans & Chiefs are powerful mechanisms for protection.
- They can be used in many other areas too, e.g. debugging, networking, migration, heterogeneous systems. Probably this list will increase with experience of the clan concept.
- Perhaps the clan concept with its hierarchy will turn out to be a usable basis for higher level concepts like distributed protection or semantic domains.

Future work has to be done, especially in the field of inter-clan migration and practical applications of the concept.

References

[1] M.Accetta, R.Baron, W.Bolosky, D.Golub, R.Rashid, A.Tevanian, M.Young: "Mach: A New Kernel Foundation for UNIX Development", in *Proc. Summer Usenix*. July, 1986.

[2] F.Baskett, J.H.Howard, J.T.Montague: "Task Communication in Demos", in *Proc. Sith Symposium on Operating System Principles*, Purdue 1977, Operating Systems Review 11,5

[3] U.Beyer, D.Heinrichs, J.Liedtke: "Dataspaces in L3", in *Proc. MIMI '88*. Barcelona, July, 1988.

[4] P.Brinch Hansen: "Operating Systems", Englewood Cliffs, 1973

[5] "EUMEL Benutzerhandbuch", University of Bielefeld. Bielefeld, 1979

[6] H.Härtig, W.Kühnhauser, W.Lux, H.Streich, G.Goos: "Structure of the BirliX Operating System", in *GMD Jahresbericht 1985*. St Augustin, 1986

[7] O.Kowalski, H.Härtig: "Protection in the BirliX Operating System", in *Proc. 10th International Conference on Distributed Computing Systems*. 1990

[8] H.C.Lauer, R.M.Needham: "On the Duality of Operating System Structures", in *Proc. Second International Symposium on Operating Systems*, IRIA, Oct. 1978, reprinted in *Operating Systems Review, 13,2*, April 1979

[9] J.Liedtke: "An Overview on the L3 Operating System", in *Proc. MIMI '88*. Barcelona, July, 1988.

[10] J.Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, G. Szalay: "Two Years of Experience with a μ-Kernel Based OS", in *Operating Systems Review 2/91*.

[11] J.Liedtke: "Clans & Chiefs – A New Kernel Level Concept for Operating Systems", GMD Tech Report No 579. St. Augustin, 1991.

[12] J.Liedtke: "Fast Interprocess Communication in the L3 Operating System", in preparation.

[13] J.Liedtke: "Task Migration Using Clans", in preparation.

[14] K.Loepere (Ed.): "Mach 3 Kernel Interface, Revision 0.5", Open Software Foundation and Carnegie Mellon University, 1990.

[15] S.J.Mullender, G.van Rossum, A.S.Tanenbaum, R.van Renesse, J.M.van Staveren: "Amoeba – A distributed operating system for the 1990s", Centrum voor Wiskunde en Informatica, Report CS-R9004, Amsterdam 1990.

[16] National Computer Security Center: "Trusted Computer System Evaluation Criteria" (Orange Book), DOD 5200.28-STD, Washington 1985

[17] R.Rashid, G.Robertson: "Accent: A communication Oriented Network Operating System Kernel", in *Proc. 8th Symposium on Operating System Principles*. December, 1981

[18] M.Young, A.Tevanian, R.Rashid, D.Golub, J.Eppinger, J.Chew, W.Bolosky, D.Black, R.Baron: "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", in *Proceeedings of the 11th Symposium on Operating System Principles*, November 1987

[19] Zentralstelle für Sicherheit in der Informationstechnik: "IT-Sicherheitskriterien" (Green Book), Bundesanzeiger Nr. 99a, Köln, 1989