

**ICE
INTARC Communication
Environment
Users Guide and
Reference Manual
Version 1.4**

Jan W. Amtrup

Universität Hamburg

Dezember 1995

Jan W. Amtrup

Arbeitsbereich Natürlichsprachliche Systeme
Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Str. 30
22527 Hamburg

Tel.: (040) 54 715 - 519

Fax: (040) 54 715 - 515

e-mail: amtrup@informatik.uni-hamburg.de

Gehört zum Antragsabschnitt: 15.4: Modulinteraktion und Hypothesenverwaltung

Das diesem Bericht zugrundeliegende Forschungsvorhaben wurde mit Mitteln des Bundesministers für Forschung und Technologie unter dem Förderkennzeichen 01 IV 101 A/O gefördert. Die Verantwortung für den Inhalt dieser Arbeit liegt bei dem Autor.

Abstract

This version of the Technical Document covers version 1.4 Patch level 0 of ICE. Previous versions are no longer supported. Changes to the previous published version of the manual are indicated by margin bars. ICE is a communication mechanism for general AI projects. It was developed at the University of Hamburg to serve as the main cooperation means between modules in an architectural study for an advanced natural language interpreting system — Verbmobil.

Besides being used for our local work, ICE has been chosen to be the communication environment for the Demonstrator and the research prototype of the Verbmobil project.

The Implementation is grounded on PVM (Parallel Virtual Machine), which is a system for communication between many processes in a heterogeneous network. On top of that we implemented an interface layer for several programming languages. It is modeled after a channel abstraction known from programming languages for parallel computers (e.g. Occam).

The goal of the implementation was to provide researchers with a simple-to-use and efficient environment for the development and testing of distributed AI-systems. Additionally we provide support for visualization by implementing an interface for Tcl/Tk, a now very en-vogue scripting language with graphics capabilities.

This paper is organized as follows:

- Chapter 1 gives you an introduction into the work with ICE. It describes briefly the different components of a system built with ICE and elaborates on the channel concept.
- Chapter 2 deals with how to obtain a copy of the software, how to install it and where to get help in case you need some assistance. Furthermore, it shows you how to get the demos running, once you retrieved ICE.
- Chapter 3 describes PVM, the basic communication software underlying ICE and the software layers ICE itself is made up of.
- Chapter 4 gives a short introduction into ILS, the Intarc License Server that is responsible for registration and informational requests.
- Chapter 5 describes the steering of ICE through the Tcl-based user interface ice-console.

- Chapter 6 describes the concept of DDL, the Data Description Language and is concerned with IDL, the Intarc DDL Layer which serves as the interface for communication using complex data structures.
- Chapter 7 contains reference information for all routines that ICE offers. All programming languages that are supported are treated there.

Contents

1	Introduction	3
1.1	Overview	3
2	Installation	8
2.1	Getting software	8
2.1.1	Getting ICE, the Intarc Communication Environment . . .	8
2.1.2	Getting PVM, the Parallel Virtual Machine	9
2.1.3	Getting Tcl/Tk	10
2.1.4	Installing ICE	10
2.2	Running demos	12
3	Communicating with ICE	13
3.1	Interference between ICE and PVM	17
4	ILS: Intarc License Server	19
4.1	Command line options	20
4.2	Configuration of ILS	20
5	The ICE Console	24
5.1	Starting the console	24
5.2	Controlling the operation	25
5.3	Using host files	26
6	IDL: Intarc DDL Layer	27
7	Reference part	28
7.1	Message tags	29
7.2	Message data types	30
7.3	Ice_Init(): Prepare for Communication	31
7.4	Ice_Release(): Return Release Information from ICE	32
7.5	Ice_Attach(): Create a component and register at ILS	34

7.6	Ice_ActiveCompo(): Information	36
7.7	Ice_Activate(): Activate a component	37
7.8	Ice_Detach(): Unregistering	38
7.9	Ice_AddCompo(): Creating a base channel	39
7.10	Ice_RemoveCompo(): Remove a component	41
7.11	Ice_AddChan(): Create an additional channel	42
7.12	Ice_RemoveChan(): Stop communicating with a partner	45
7.13	Ice_Send(): Send a message	46
7.14	Ice_Broadcast(): Send a message to all components	49
7.15	Ice_GetSentModuleId(): Retrieve module Id	52
7.16	Ice_SetTurnId(): Set turn id for following messages	53
7.17	Ice_Receive(): Get a message from other components	54
7.18	Ice_TReceive(): Receiving with timeout	56
7.19	Ice_Probe(): Check for incoming messages	58
7.20	Ice_SetMinTurnId(): Set the minimum turn id for following receives	60
7.21	Access functions for message data	61
7.22	Idl_UserInit(): Initialization of IDL	63
7.23	Idl_Free(): Deallocate memory	64
7.24	Ice_Handler(): Event-based message handler	65
7.25	Ice_Handler(): Event-based message handler: Widget commands	67

Bibliography	69
---------------------	-----------

Chapter 1

Introduction

1.1 Overview

The *Intarc Communication Environment* ICE is a software environment for the development of distributed AI-systems¹. It is designed to give a flexible and efficient communication means that is oriented at an abstract model suitable for such purposes. It uses a publicly available communication software (PVM) that is widely used and realizes interface for several programming languages used in AI.

The global architecture of a system designed and constructed using ICE is shown in fig. 1.1². A system consists of several components which may be written in any programming language. Each component gets a unique name and may communicate with all other components of the system. A special component (ILS, *Intarc License Server*) acts as a “conferencier” which introduces new components to the system and handles informational requests.

After having attached to the ILS, components communicate with one another using *channels*.

Of the most common available communication methods, namely *Shared memory*, *Remote procedure calls* and *Message passing* we assume the last one to be the most appropriate for distributed AI-systems. One doesn’t run into memory contention problems when working with many components on the same shared memory segment and doesn’t get access right problems. Remote procedures work synchronously and eventually produce waiting periods. Message passing on the other hand lends itself easily to a task-oriented system that asynchronously issues requests and answers requests from other components.

¹Several people contributed to this system by providing ideas and hints, especially Marcus Kessler and Andreas Hauenstein.

²See also (Amtrup, 1994)

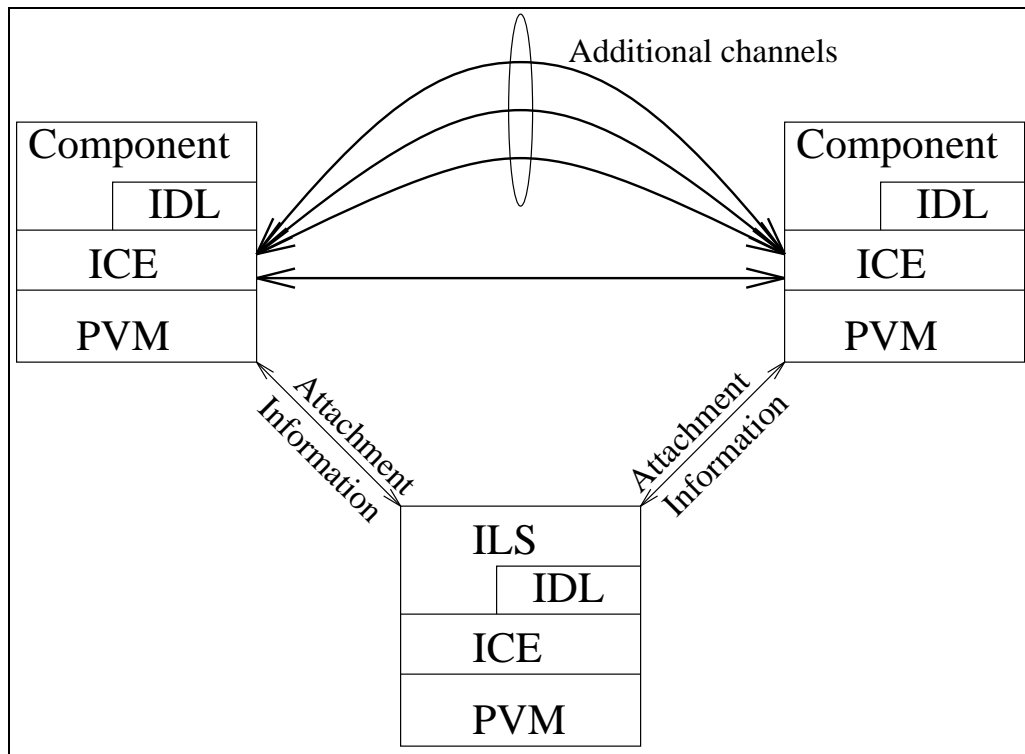
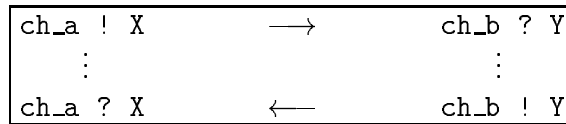


Figure 1.1: The overall structure of an ICE-system

Channels are abstract communication paths along which messages are transported. The theoretical roots stem from Hoare (1978). The most widely known implementation of this model is the Inmos Transputer which is equipped with hardware links that implement channels and their protocols on the microcode level (Graham and King, 1990). Additionally the programming language Occam (see, e.g., Burns (1988)), directly operates on this channel model. These channels are unbuffered point-to-point communication links where messages are sent and received synchronously using Rendez-Vous synchronization:



ICE uses a channel abstraction, too. The main differences lie in the support of a large set of hardware architectures on which PVM operates and the possibility to configure certain channel characteristics. ICE offers a *base channel* between every two components. This channel works asynchronously and uses the *eXternal Data Representation* XDR (Corbin, 1990) to encode data hardware-independent. For the base channel there is no control of “fairness” of certain components, they may send as many messages as they want. This base channel is constructed when the first communication between two components starts and exists until the termination of the system.

Besides this base channel there may exist several additional channels that are constructed upon request from both communication partners. These channels can be configured to fit certain needs:

- Synchronous operation. The sender waits until a message is completely delivered before work is started again.
- Loss of encoding. A pair of components may decide to use a raw data channel that is not encoded. This allows faster message exchange, but is only possible if it is known that both partners have the same hardware architecture. A Broadcasting of messages (that is, a simultaneous sending to all components in the ICE system) is only possible on the base channel.
- Tokenized channels. Analogous to some LAN architectures one can decide to introduce a token on an additional channel. Again, this is not possible for base channels. Each tokenized channel is equipped with its own token, so there is no interference between different channels. The impact of a token is that only one communication partner is allowed to send. The ownership of the token licenses the right to send messages. A component can actively

give the token away. In case it wants to send it may wait for the token to arrive but a component can not be forced to give its token away. The idea behind the introduction of a token is to eliminate the situation of both components sending messages and never accepting any.

The communication primitives of ICE are based on PVM, the Parallel Virtual Machine (Geist et al., 1994) from Oak Ridge National Laboratory. It is an implementation which allows the distribution of a problem onto a heterogeneous computer network: most of the common Unix-workstation are supported as well as some parallel computers like the Connection machine. PVM even runs on a PC equipped with Linux. It offers a widely used, stable and efficient communication means that is suitable for extensions like the one we did.

PVM offers routines to en- and decode basic data types in order to communicate them. Together with an extension (IDL: Intarc DDL Layer) it is possible to send and receive certain types of complex data structures transparently over a network and across language boundaries. This extension is optional, as shown in fig. 1.1. If it is omitted, only scalar data types can be transmitted.

PVM is steered by means of an X-based console, which allows issuing of commands to add machines and tasks to the virtual machine and offers a small amount of debugging. This console is actually not used by ICE and has only informational importance.

At the moment, ICE supports six languages with different dialects:

- C
- C++
- Allegro Common Lisp
- CLISP
- LUCID Common Lisp
- Sicstus Prolog
- Quintus Prolog
- Tcl/Tk

A code fragment cut out of a demonstration program for C may serve as a first guideline how to program with ICE:

```

/* Initialization */
Ice_Init();

/* create component and register at ILS */
me = Ice_Attach( "demo-sender");

/* create other component */
dest = Ice_AddCompo( "demo-receiver");

/* send ordinary message of various types to destination task */
demo_string = "Howdy";
Ice_Send( dest, ICE_NOTAG, 1, "", 0, 0, 50, IDL_STRING, demo_string);

/* receive messages from destination task */
Ice_Receive( ICE_ANYCHAN, ICE_ANY, &msg);
printf( "%s \n", (char *)msg.m_Data);

/* clean up */
Ice_RemoveCompo( dest);
Ice_Detach( me);

```

This fragment establishes a connection between the components *demo-sender* and *demo-receiver*. It then sends a message “Howdy” along that channel and waits for an answer. Finally, the communication devices are shut down.

The forthcoming chapters describe in depth how you can obtain ICE and the various pieces of software needed for the setup. Each routine is described for every programming language supported.

Chapter 2

Installation

ICE was written as part of the German joint research project Verbmobil. It is therefore freely available for all researchers within Verbmobil. All others should direct a request to us to obtain a copy.

2.1 Getting software

2.1.1 Getting ICE, the Intarc Communication Environment

ICE can be ftp'ed from the German Verbmobil server using a script as below:

```
nats13% ftp ftp.dfki.uni-sb.de
Connected to ftp.dfki.uni-sb.de.
220 com-serv FTP server (Version wu-2.4(1) ...) ready.
Name (ftp.dfki.uni-sb.de:jan): VM-depot
331 Password required for VM-depot.
Password:
230>Welcome to the Verbmobil ftp server.
230-
230 User VM-depot logged in. Access restrictions apply.
ftp> cd FTP-SERVER/vm-module
250 CWD command successful.
ftp> bin
200 Type set to I.
ftp> get ice_v1.4.tar.gz
ftp> quit
```

Note that the account `VM-depot` is protected via a password. If you don't have access to this server or run into problems retrieving the software, send mail to `ice@nats2.informatik.uni-hamburg.de` and we will send you a copy.

2.1.2 Getting PVM, the Parallel Virtual Machine

The least thing you need besides ICE is PVM, the Parallel Virtual Machine from Oak Ridge National Laboratory. You may get it through anonymous ftp using the following script:

```
nats13% ftp netlib2.cs.utk.edu
Connected to netlib2.cs.utk.edu.
220 netlib2 FTP server (Version ...) ready.
Name (netlib2.cs.utk.edu:jan): anonymous
331 Guest login ok, send your complete e-mail address as password.
Password:
230-Welcome to ftp.netlib.org
230 Guest login ok, access restrictions apply.
ftp> cd pvm3
250 CWD command successful.
ftp> bin
200 Type set to I.
ftp> get pvm3.3.10.tar.gz
ftp> get ug.ps
ftp> quit
```

PVM 3.3.10 is the most actual version; we require at least PVM 3.3.4 to be present. Note that you **MUST** set the environment variable `PVM_ARCH` manually to `SUN4SOL2` if the automatic identification process returns `SUNMP`. This is due to a bug in Allegro concerning the use of the Threads-Library. Franz is working on it (I think, for just over half a year now), but ...

Also note that you have to edit the file `pvm3/conf/HPPA.def` if you are using a HP machine, compile using the HP C-compiler, and want to run quintus programs. Edit the line describing the architecture dependent C Flags to contain `+Z` which denotes position-independent code with the HP C compiler. And don't get impatient if the loading of the foreign function files for Allegro takes several minutes to complete (~30 min in my experience). And don't ask me why this is, either.

Note also, that you can't run Lucid components if you compile this way. Lucid requires — for what reason ever — compilation without `+Z`. Thus, if you want

to run both Allegro and Lucid on a HP, you have to install two versions of the PVM libraries and the ICE library.

The file `ug.ps` is the actual Users guide for PVM, which may help at certain points. If you want, you may download the graphical interface for PVM, which is located in the directory `xpvm`. If you like, you may have a look at the URL <http://www.netlib.org/pvm3/book/pvm-book.html> with Netscape or one of its friends. There is also a newsgroup `comp.parallel.pvm` which is deserved especially to PVM.

2.1.3 Getting Tcl/Tk

Tcl and Tk are optional. You don't need them to work with ICE. But if you want to add graphic support to your application, you might find it helpful. There are a number of anonymous ftp Server where you can get a copy of Tcl/Tk nowadays. If you don't find one, try

```
ftp://ftp.informatik.uni-hamburg.de/pub/lang/tcl. You will need at least the versions Tcl 7.3 and Tk 3.6. The newsgroup for Tcl is comp.lang.tcl.
```

The ICE console now runs with Tcl 7.4 and Tk 4.0. In some environments, these versions prevent the console from hanging while looking for ILS messages, so you might prefer using the new versions of Tcl and Tk.

2.1.4 Installing ICE

You will find the installation procedures for PVM and Tcl/Tk in the respective README-files or the manuals. Regarding ICE, there is a README-File with roughly the same content as what you read now. In case of divergences, the README-file should be more actual.

First you have to choose a directory where ICE should reside and unpack the distribution archive there:

```
nats13% mkdir ice
nats13% cd ice
nats13% gzip -cd ice_v1.4.tar.gz | tar xf -
```

Now you should either establish a link to one of the predefined definition files or copy the `Definitions.generic` file and edit its content to suit your environment. We provide Definitions files for various hardware settings (and in some cases even for different sites within Verbmobil...).

Make sure that the environment variables `PVM_ROOT` and `PVM_ARCH` are set accordingly, otherwise something will go wrong. The `PATH`-Variable should contain references to the `pvm3` directory. The environment variable `ICE_ROOT` should

point into the earlier created directory. Your initialization file could (as an example) contain

```
setenv PVM_ROOT /opt/pvm3
setenv PVM_ARCH '$PVM_ROOT/lib/pvmgetarch'
set path=($path $PVM_ROOT/lib/$PVM_ARCH)
setenv ICE_ROOT ~/ice
```

Note that if PVM_ARCH is set to SUNMP this way, you MUST specify

```
setenv PVM_ARCH SUN4SOL2
```

if you want to use Allegro Common Lisp. At this point, you can build ICE by calling

```
nats13% make depend
nats13% make
nats13% make install
```

It works best with the Gnu compilers g++ and gcc (in the version 2.7.0 or later). Version 2.5.8 is supported, too. There are differences concerning the handling of template definitions. I personally use the Gnu-version of make, **gmake**, but a normal make should do as well. Ignore all warnings concerning directories that already exist.

On my machine, makedepend complained about a missing `_IO_config.h`. I created it in the GNU library directory by simply touching it. Ask your system administrator to check that file if you don't have the necessary permissions.

PVM searches some of its programs in the home directory of the user. This is normally no problem since the only program to be searched is the PVM group server (PVMGS) which you can start manually (it is located in the PVM directory under `$PVM_ROOT/bin/$PVM_ARCH`, but in heterogeneous environments (eventually with shared home directories) it helps to create directories for every architecture you are running and create a link for the group server like this:

```
nats13% mkdir ~/pvm3
nats13% mkdir ~/pvm3/bin
nats13% mkdir ~/pvm3/bin/$PVM_ARCH
nats13% cd ~/pvm3/bin/$PVM_ARCH
nats13% ln -s $PVM_ROOT/bin/$PVM_ARCH/pvmgs
```

2.2 Running demos

The executables for the demonstration programs are built during the installation process just described. All you need to do now is start ICE and let the demo programs run. In version 1.3 and later this is simply done by starting the ICE console and pressing the “Start ICE” button.

```
nats13% $ICE_ROOT/bin/ice-console &
```

Now open up two terminal windows and start a demonstration program in each of them. They should give you some output in which case you can safely assume that you got ICE running at last.

```
nats13a:% $ICE_ROOT/bin/$PVM_ARCH/c-demo send  
nats13b:% $ICE_ROOT/bin/$PVM_ARCH/c++demo receive
```

After operation you may want to stop PVM running. This is done by pressing the button “Stop ICE” on the ICE console.

Chapter 3

Communicating with ICE

ICE builds up the interface between components of a distributed application and PVM which realizes the communication physically. It offers data structures and functions for the definition of components, channels and messages and their communicative behavior. By using ILS the components can address themselves by names. The functions may be grouped together as such:

- **Initializing, Attaching, Detaching.** These functions introduce a component to the distributed system and ensure that all preliminary tasks are done. After completing the work of the system, they guarantee a normal shutdown operation.
- **Maintaining base channels.** Upon declaring a component, a base channel on which basic communication with the new component should happen is announced. It is actually created when communication is encountered for the first time. It is good practice to remove a base channel before detaching from the whole system.
- **Maintaining additional channels.** As elaborated in chapter 1, the channel model of ICE offers additional channels which can be configured to fit ones needs. So, an analogous set of routines exists for these channels.
- **Sending and Receiving.** There are some functions for the real work to be done. As parameters they expect a more or less complete description of what is being sent or received. The following terms are used:
 - **Turn Id.** As ICE is developed to serve in systems designed for dialogue interpreting, a turn is a crucial notion denoting a single contribution of one speaker. Turns get numbered with integers, starting from 1. Every component is responsible to supply the sending functions with the id

- of the turn that message belongs to. Whenever a message is received, the turn id gets included into the message structure returned.¹
- Concern. You can specify a concern for each message you send to another component. This makes tracking of messages belonging to a decent context easier (e.g in case of interactive systems capable of question answering). The concern is a string with no internal semantics.²
 - Module id. Every message that is treated within the ICE system gets a unique identifier represented as a string containing the name of the module sending that message and an module-internal counter. This identification can be used for reason maintenance tasks applying to messages.³
 - Message tag. Every message gets a label that identifies the purpose of it. There are some tags that are used for maintenance purposes (e.g. `ICE_WHO`), others simply serve as a hint that no tag is to be used (`ICE_NOTAG`). Message tags can be used to partition the set of messages on the base band into logically connected subsets; thus one could decide to use different tags for data and control information. We would rather propose to use additional channels for this purpose. You can't use different message tags on additional channels, though (the reason for this being that additional channels are emulated using message tags ...).
 - Start time, end time. ICE was developed for the architectural prototype of Verbmobil, INTARC. The information about the interval in time a hypotheses spans is a prominent information herein. So we decided to establish special fields for the starting point and the end point of each message. This allows future releases of ICE to take a closer look at such information. Times are normally measured in cycles of 100 ns.
 - Priority. The same reason led us to incorporate the priority or rating into the header information of each message. We assume priorities lying in $[0 \dots 100]$, but no other restrictions apply.
 - Data type. ICE is capable of transferring several data types. Each data type is determined by use of a separate tag. Standardly, all scalar data types including strings are supported. If you choose to use IDL, the Intarc DDL Layer, you may define a restricted set of complex

¹The turn id was introduced with version 1.3.

²The concern was introduced with version 1.3.

³The module id was introduced with version 1.3.

data types and use them transparently. Chapter 6 explains the tags to be used for the basic data types as well as the definition of more complex ones.

Normally, one calls some kind of `Ice_Send()` and `Ice_Receive()` to use communication. Besides this, a distribution of a message to all components known to work in the actual system can be sent by using a broadcast call. Using `Ice_Probe()`, a component can decide whether a message is actually present and could be delivered by calling the receiving routine.

In some languages, there is the need to release allocated memory space. ICE offers functions that carry out this task for every known data type.

- Configuration and Housekeeping. With version 1.3, there are now some configuration options that can be set using the appropriate function. Additionally, you can specify the minimum turn id a component is interested in as well as the turn id to be used in further send calls.

Table 3.1 summarizes the routines and their purpose. Further information is contained in the reference part of this manual, chapter 7.

Routine	Description
Ice_Init() Idl_UserInit() Ice_Release() Ice_MajorRelease() Ice_MinorRelease() Ice_Attach() Ice_Activate()	Initialize basic ICE system Initialize functions for user-defined data types Return Release Information Attach and name component Activate a different component which has been attached previously
Ice_ActiveCompo() Ice_Detach()	Return the active component Remove connection to rest of ICE system
Ice_AddCompo() Ice_RemoveCompo()	Declare a component to communicate with Clean up data for component
Ice_AddChan() Ice_RemoveChan()	Establish an additional channel to an already known component Remove such a channel
Ice_Send() Ice_Broadcast() Ice_GetSentModuleId() Ice_Receive() Ice_Probe() Ice_Free() Ice_SetMinTurnId() Ice_SetTurnId() Ice_Configure()	Send a message to a destination component an a channel Send a message to all other components on the base channel Retrieve the module id of the last message sent. Receive messages on certain channels with certain tags Check to see if a message is available Clean up memory used for data structures Set the minimal turn id this component is interested in. Set the actual turn id for the following messages. This function is only present where abbreviated function calls are available. Set various configuration options

Table 3.1: Overview of the routines of ICE

3.1 Interference between ICE and PVM

There is some interference between ICE and PVM. Mainly, this is due to the addressing of components through names which are converted to task ids by the appropriate ICE routines. Table 3.2 summarizes which Routines can be safely used and which may not.

Function	Ok?	Note	Function	Ok?	Note
pvm_addhosts	Ok		pvm_notify	Ok	5
pvm_barrier	Ok	1	pvm_nrecv	-	2
pvm_bcast	-		pvm_pk*	-	2
pvm_bufinfo	-	2	pvm_parent	Ok	
pvm_catchout	Ok		pvm_perror	Ok	
pvm_config	Ok		pvm_precv	-	
pvm_delhosts	Ok		pvm_probe	-	2
pvm_exit	Ok		pvm_psend	-	
pvm_freebuf	-		pvm_pstat	Ok	
pvm_getinst	Ok	3	pvm_rcv	-	2
pvm_getopt	Ok		pvm_recvf	-	2, 6
pvm_getrbuf	-		pvm_reduce	Ok	3
pvm_getsbuf	-		pvm_reg_hoster	Ok	
pvm_gettid	Ok	3	pvm_reg_rm	Ok	
pvm_gsize	Ok	3	pvm_reg_tasker	Ok	
pvm_halt	Ok		pvm_send	-	3
pvm_hostsync	Ok		pvm_sendsig	Ok	4
pvm_initsend	-	2	pvm_setopt	Ok	
pvm_joingroup	Ok	3	pvm_setrbuf	-	
pvm_kill	Ok	4	pvm_setsbuf	-	
pvm_lfgroup	Ok	3	pvm_spwan	Ok	
pvm_mcast	-		pvm_tasks	Ok	
pvm_mkbuf	-		pvm_tidtohost	Ok	4
pvm_mstat	Ok		pvm_trecv	-	2
pvm_mytid	Ok		pvm_upk*	-	2

Table 3.2: PVM routines and their use together with ICE

The following notes apply:

- 1 You should be aware of side effects. If one component handles a receiving call while another issues a `pvm_barrier()` it may never be able to do so, too. in that case, the application would get stuck.

- 2 ICE uses this functions for sending and receiving messages. You should not intervene with these, except if you specify packing and unpacking routines for special data types.
- 3 The current version of ICE uses only one group with one single member, the ILS. You can configure the name of the group used (which defaults to "INTARC"). You must not use any group used for an ICE application in the virtual machine running for any other purposes.
- 4 You have to provide a task id to this call. For ICE components that are already known to your component, experienced C programmer can use `Icep_GetCompoByName(char *name)->tid` but this could be 0 in case the ILS has not acknowledged the presence of the other component.
- 5 Note that the ILS uses notification on `PvmTaskExit`.
- 6 `pvm_recvf()` is crucial to message selection for ICE. During a receive or probe, the selection function is always set up before calling `recv()` etc. and immediately restored.

Chapter 4

ILS: Intarc License Server

The ILS serves as a central reception for components. All components that enter the system have to drop a note what their purpose is. The ILS has the complete information about what components are actually in the system and can answer requests for addresses of certain components. Furthermore, the ILS offers the possibility to establish system-wide services an application could benefit from.

No component ever communicates overtly with the ILS, this is done inside the appropriate ICE-functions. For sake of completeness we describe here the main functions of the ILS:

- Every component that is started registers at the ILS. This is done with a `ILS_ADD`-message. The method works asynchronously, no replies from ILS are necessary. The attachment is done by the function `Ice_Attach()`.
- Every time a component ends operation, it should unregister at the ILS. This is implicitly done by calling `Ice_Detach()` which in turn sends a `ILS_DEL`-message to the ILS.
- The ILS stores a table of all components taking part in the actual computation. It can thus answer requests from components that want to know the PVM-Address of certain other components. Again, this is done implicitly when needed. If, for example, an `Ice_Send()` is called for the first time and the address of the destination is not known, the component waits until a notification from the ILS arrives with the necessary task id. The ILS keeps track of which component may communicate with what other and distributes `ILS_WHO`-messages when appropriate.

The component can control its behavior in this case. Issuing a `Ice_Configure(NOBLKSEND, 0, "")` call, the sending functions return with an error indicator when a component is not yet known.

Components within a system can be started in any order, but both endpoints of a channel have to be started and attached before communication over that channel can be successfully done.

4.1 Command line options

With version 1.2, ILS offers two command line options:

- v Verbose setting. If provided in the command line, ILS issues messages about what it is doing. All requests from components and the actions taken by the ILS are given in a protocol.
- f Configuration. This command line option specifies the name of a configuration file to be used for this run. See below for further details.
- g Group. This option allows you to start several ICE application within the same virtual PVM machine. The ILS enters a group to be identifiable by ordinary components that have to attach at the ILS. Using the **-g** option, you can specify the group name this instance of ILS should use. Additionally, you must issue a call to `Ice_Configure(ILSGROUP, 0, groupName)` to configure your application to use the same group. The default group used by the ILS is `INTARC`.
- s Secure. This option specifies that the ILS shall wait a short time before distributing messages notifying components of the destruction of a channel of the absence of a component. This can be helpful in case of races where e.g. detachment notifications appear at the destination component of a message before the last message from the disappearing component has been received. note that this option can nevertheless not guarantee the successful delivery.

4.2 Configuration of ILS

Version 1.2 and following offer the possibility to configure channels transparently for the components. This facility was requested in order to be able to split up data paths between components¹. Two main reasons for this behavior were:

¹Split channels were suggested by Marko Auerswald.

Config-file	::= Config-spec ⁺
Config-spec	::= Channel-spec Real-spec ⁺ {blank line}
Channel-spec	::= Source Destination Channelname
Real-spec	::= Real-destination Real-channel Flags Send? Receive?

Figure 4.1: BNF for configuration files

- Integration of a (graphical) user interface for parts of the system. One should be able to listen on a channel. This is done by splitting up the sending side and let a channel deliver its messages to more than one recipient.
- Insertion of Filters, Compilers for interface specifications etc. If necessary, one wants to cut a channel in two pieces and insert a mechanism to change the data sent over that channel. This could be useful if one has to maintain a complex systems with interchangeable parts. If two subsystems offering the same functionality only differ in behalf of their interface specification, it might be easier to insert a compiler for the data along that interface than to modify the component's code.

Both extensions have to be transparent for the affected components. We introduce a configuration file option for the ILS (command line option `-f`, see above). The file named with that option has to follow the syntax given in Fig. 4.1.

Channels are conceptually split up into two ends (one at each side of the channel). You have to describe both ends of the channel within the configuration file. Each side consists of a list of real channels. Each real channel points to a component, has a name and some flags and is labelled to be used for sending messages or for receiving messages. Any number of sending real channels and any number of receiving real channels can be configured.

Fig. 4.2 shows an example configuration. Two components, A and B, are connected with a channel depicted by a dashed line. The messages sent by both components are visualized in two further components, `UIG_A` and `UIG_B`. Component C is a mapper for interface data and changes data sent from A to B. The reverse direction (B to A) is handled directly.

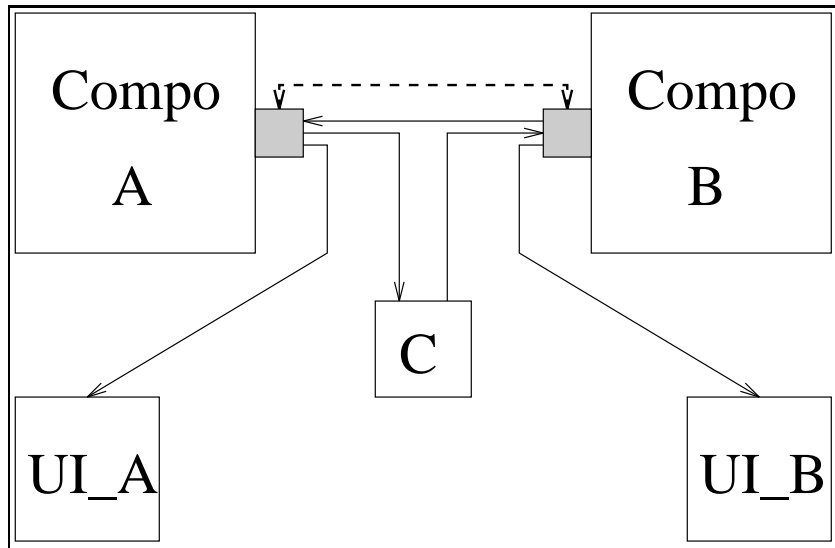


Figure 4.2: Configuration example

The configuration describing this situation might look like this:

```
A B BASE
UIG_A BASE -1 1 0
C BASE -1 1 0
B BASE -1 0 1
```

```
B A BASE
UIG_B BASE -1 1 0
A BASE -1 1 0
C BASE -1 0 1
```

Note the following details:

- Both split channel definitions have to be present.
- Although all channels are basic ones in the present example, you may use additional channels as well.
- The flags given are always -1 (corresponds to ICE_NORMAL).
- Sending and Receiving on real channels is controlled via entry of 0/1 in the corresponding columns.

- In case you specify multiple sending and receiving channels, you should be aware that minor accidents in the coding of communication handling may result in doubly processed messages.

Chapter 5

The ICE Console

It is boring and error prone to start and stop the ICE system by typing in shell commands and commands of the PVM console. Even the scripts provided with earlier versions of ICE were no great help if you used configurations which involved more than one machine. This made us construct a graphical user interface with which the steering of ICE should not be a great deal anymore. This chapter explains the ICE console and their use.

5.1 Starting the console

The console is started by issuing the command

```
${ICE_ROOT}/bin/ice-console [yes|no [<hostfile-name>]]
```

Typically, you want to start the console in the background. The parameters to the command have the following meaning:

- The first (boolean) parameter specifies whether or not ICE should be started automatically after setting up the user interface. This saves a mouse click and allows the construction of automatically running systems from a script. The default value is `no` meaning that ICE is not started after loading the console.
- The second parameter specifies the name of the host file to be used initially. If no name is provided, the console assumes that PVM is to run on the local machine only.

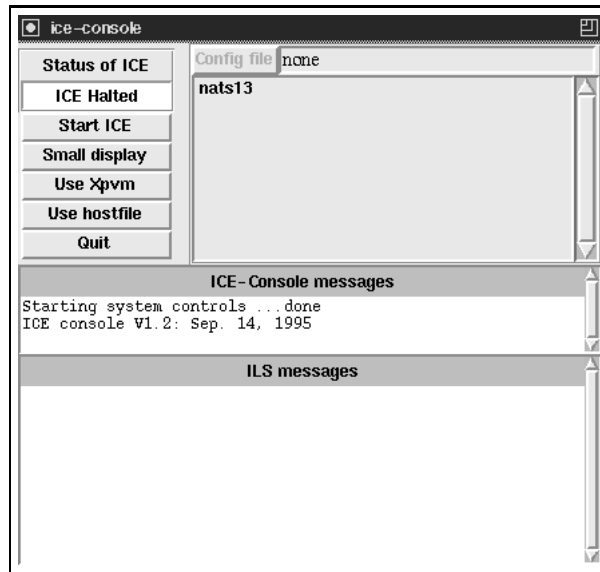


Figure 5.1: The console after startup

5.2 Controlling the operation

After calling the console, you should see a window on your screen looking very much like Fig. 5.1. It is divided into four parts:

- The left upper corner contains some buttons and a status information. The status may have two values: “ICE Halted”, which is indicated in black on white, and “ICE running”, which is written white on red to draw your attention to that fact. The button directly beneath the status line swaps between the two modes.

The button “Small display” lets the window shrink by hiding the information about the configuration file and the messages generated by the console and the ILS. It changes to “Large display” to give a means to recover this information.

“Use XPVM” affects the startup sequence of ICE in a way that an instance of XPVM, the graphical user interface to PVM is started besides the pure PVM daemon and the ILS. That way, you have access to information about every single message in the system. Note that you have to compile ICE with the setting “DEBUGGING = -DXDEBUG” specified in $\${ICE_ROOT}/Definitions$ in order to get information from components which were not started man-

ually within XPVM. This button is again a two-valued one, toggling the actual status of XPVM usage. Note that you can only change the behavior if ICE is currently not running.

“Use hostfile” (again a toggling button, the opposite direction being “Don’t use hostfile”) specifies whether PVM is started on a predetermined list of hosts or just on the host where the console is running. See below for more information on the use of host files. This button is only operational if ICE is not running.

“Quit” does what it is supposed to. A running ICE session is ended prior to leaving the console.

- The right upper corner defines the name and content of the hostfile if a hostfile is used. The button “Config file”, which is only active if a hostfile is in use, lets you select a different hostfile for the next session of ICE running. The list below the button contains an entry for each machine that is part of the virtual machine. The name of the machine along with any additional parameters specified in the hostfile is displayed. Note that you cannot edit within the list.
- the middle part of the console window is reserved for messages from the console. Mainly this are messages about starting and stopping the system.
- The bottom part of the window contains messages from the ILS for the actual run. It is cleared with each new invocation of ICE. The ILS is started with the command line option `-v` to retrieve those messages. This is useful in case you run into problems with ICE and want to analyze what has happened.

5.3 Using host files

Host files are very useful if you want to run your application on several machines in the local net. PVM supports the use of hostfiles when starting PVM through the PVM console. ICE adopts this behavior. If a host file is specified, a PVM daemon is run on each machine specified in the host file. The syntax is exactly as defined in the PVM book (Geist et al., 1994), that is, one machine per line together with possibly several options (different user name etc.).

Upon startup, the console assumes that you want to start ICE only on the local machine. You can alter this by specifying a host file in the command line (see above).

Chapter 6

IDL: Intarc DDL Layer

The Data Description Language DDL is a formal language that allows definition of reasonable complex data types which in turn can be transparently communicated using IDL, the Intarc DDL Layer. The concept of DDL is described elsewhere (Pyka, 1992), so we can concentrate on the usage here.

This part has to be elaborated. You should read about defining new data types, compiling them into interface specifications for the various programming languages and using them by calling functions with specified data type tags. As it happens, the compilers are not ready right now. What's more, we didn't decide how to represent complex data structures in some languages. So you have to wait...

DDL and IDL are completely optional. You can use ICE safely and efficiently without them. Of course, you can't send and receive user-defined complex data structures, but the scalar data types are supported.

Chapter 7

Reference part

This chapter contains reference information for the use of all routines ICE knows. The first parts of this chapter define message tags and scalar data types, the following sections each describe one logical function for all programming languages part of ICE.

7.1 Message tags

Table 7.1 shows the message classes ICE knows. Tags that are tagged „internal“ are only used by ICE and must not be used by you. The „name“-column shows the different forms of the tag for the different programming languages vertically. Usage defines whether the tag can be used for Sending (S) and/or Receiving (R).

Kind	Name (C, C++) Lisp Prolog	Usage	Description
	ICE_ANY +ICE_ANY+ ice_any	R	Under-specified. Listen to messages of any task
	ICE_CMPTAG +ICE_CMPTAG+ ice_cmptag	R	under-specified. Listen to messages of this task, disregarding message tags
	ICE_NOTAG +ICE_NOTAG+ ice_notag	S/R	Used for messages on additional channels that cannot be tagged. Used as a standard tag on the base band.
Intern	ILS_ADD		Register at ILS
Intern	ILS_DEL		Unregister at ILS
Intern	ILS_WHO		Notification about the task id of a component from ILS
Intern	ILS_CHC		Notification to create a channel
intern	ILS_CHS		Notification about a sending real channel
Intern	ILS_CHR		Notification about a receiving real channel
Intern	ILS_CHE		Notification about the completion of the configuration of a channel
Intern	ILS_CHD		Request and notification about the deletion of a channel
Intern	ILS_TASKEEXIT		Notification for ILS that one component left disgracefully

Table 7.1: Message classes

7.2 Message data types

The data types that are defined within ICE are shown in table 7.2. Only scalar data types are incorporated that can be communicated without IDL. The corresponding types are shown for each programming language. Examples can be found under the definitions.

Depending on the internal representation of data within the respective programming languages there may be some conversion difficulties. This affects mostly the representation of integers. However, it should be safe to treat integers with IDL_INT as long as they are within the range $-2^{29} \leq x \leq 2^{29} - 1$ — this range is for Sun4 with Solaris 2.3.

Name (C/C++)	C/C++	Lisp	Prolog	Tcl	Description
Lisp Prolog					
IDL_CHAR +IDL_CHAR+ idl_char	char 'a'	character #\a	atom a	string "a"	One character
IDL_STRING +IDL_STRING+ idl_string	char * "Howdy"	string "Howdy"	atom 'Howdy'	string "Howdy"	Character string
IDL_INT +IDL_INT+ idl_int	int -4711	fixnum -4711	integer -4711	string "-4711"	Integer
IDL_UINT +IDL_UINT+ idl_uint	uint 4711	fixnum 4711	integer 4711	string "4711"	Unsigned integer
IDL_LONG +IDL_LONG+ idl_long	long -123456	fixnum -123456	integer -123456	string "-123456"	Long integer
IDL_ULONG +IDL_ULONG+ idl_ulong	ulong 123456	fixnum 123456	integer 123456	string "123456"	Unsigned long integer
IDL_FLOAT +IDL_FLOAT+ idl_float	float 3.4	single-float 3.4	float 3.4	string "3.4"	Floating point number
IDL_DOUBLE +IDL_DOUBLE+ idl_double	double 1.33	double-float 1.33	float 1.33	string "1.33"	Double precision floating point number
— — idl_term			term [a, p(b, c, [])]		Arbitrary Prolog term

Table 7.2: Message data types supported by ICE

7.3 Ice_Init(): Prepare for Communication

This function declares all parameters for communication and enters some data into the internal hash tables. It has to be called before any other function of ICE is used.

Synopsis

```

C          void Ice_Init( void);
C++       --
Lisp      --
Prolog    ice_Init.
Tcl       --

```

Parameters

None.

Discussion

Initialization is done implicitly for C++, Lisp and Tcl. It happens upon creation of the first component.

Lisp and Prolog components have to load the component code prior to using ICE. See the example components.

If you use data types specified with DDL you have to use `Idl_UserInit` instead.

Example

```

C          Ice_Init();
C++       --
Lisp      --
Prolog    ice_Init.
Tcl       --

```

Errors

None.

7.4 Ice_Release(): Return Release Information from ICE

The functions `Ice_Release()`, `Ice_MajorRelease()` and `Ice_MinorRelease()` provide information about the version of ICE running. They can be used to check a certain minimum version without which some extensions would not run.

Synopsis

```

C      char *release = Ice_Release( void);
      char *majorRelease = Ice_MajorRelease( void);
      char *minorRelease = Ice_MinorRelease( void);
C++   char *release = Ice_Release( void);
      char *majorRelease = Ice_MajorRelease( void);
      char *minorRelease = Ice_MinorRelease( void);
Lisp   release-string ←— (release)
      major-release-string ←— (major-release)
      minor-release-string ←— (minor-release)
Prolog ice_Release( -release).
      ice_MajorRelease( -major_release).
      ice_MinorRelease( -minor_release).
Tcl    release ←— Ice_Release
      majorrelease ←— Ice_MajorRelease
      minorrelease ←— Ice_MinorRelease

```

Parameters

- `release` – String. Upon return, it contains a string describing the version of ICE you have at hand (eg. "ICE, Version 1.3p10 of 25 Sep 1995").
- `majorRelease` – String. Upon return, it contains a string describing the major release number of ICE (e.g. "1")
- `minorRelease` – String. Upon return, it contains a string describing the minor release number of ICE (e.g. "3")

Discussion

Example

```
C      release = Ice_Release();
      majorRelease = Ice_MajorRelease();
      minorRelease = Ice_MinorRelease();

C++   release = Ice_Release();
      majorRelease = Ice_MajorRelease();
      minorRelease = Ice_MinorRelease();

Lisp  (setf current-release (release))
      (setf current-major-release (major-release))
      (setf current-minor-release (minor-release))

Prolog ice_Release( Release).
      ice_MajorRelease( MajorRelease).
      ice_MinorRelease( MinorRelease).

Tcl   set release [Ice_Release]
      set majorrelease [Ice_MajorRelease]
      set minorrelease [Ice_MinorRelease]
```

Errors

None.

7.5 Ice_Attach(): Create a component and register at ILS

To take part in communication with ICE, a component has to create a structure that holds critical information. Also, the connection with the ILS, the Intarc License Server, has to be done before messages can be sent or received. `Ice_Attach()` does this during the first attachment. Note that you may specify the group the ILS is using with `Ice_Configure()`. This has to be done before the first attachment takes place.

Synopsis

```

C           Ice_Compo *compo = Ice_Attach( char *name);
C++        Component::Component compo( char *name);
Lisp       compo ← (attach name)
Prolog     ice_Attach( +name, -Compo).
Tcl        compo ← Ice_Component name
           compo attach
    
```

Parameters

- `name` - String. It contains the name of the component to be attached.
- `compo` - Return value. The routine builds up a structure or object that describes the attached component. The component need not care about the actual content.

Discussion

For C++ and Lisp this routine works as a constructor for objects of the appropriate type. In case of Prolog a pointer to a structure is returned that acts as a placeholder for the component. The Tcl implementation consists of two commands: The first one creates a object that is named after the component and can be used for communication, the second one attaches the component to ILS.

You may issue any number of calls to `Ice_Attach()`. Many components become part of the system. You can thus handle different components within one memory image. This is useful to integrate components neatly. The functions `Ice_ActiveCompo()` and `Ice_Activate()` handle the selection of the currently active component.

At every time, there is one active component that may send and receive messages and which is used as reference in all other ICE-related calls. `Ice_Attach()` also switches the active component to be the one just created.

Example

```
C      Ice_compo *me;
      me = Ice_Attach( "parser");
C++    Component me("parser");
Lisp   (setf me (attach "parser"))
Prolog ice_Attach( 'parser' , Me).
Tcl    Ice_Component parser
      parser attach
```

Errors

None.

7.6 Ice_ActiveCompo(): Information

This function returns the component now active.

Synopsis

C	<code>Ice_Compo *Ice_ActiveCompo(void);</code>
C++	<code>int Component::Active Component(void);</code>
Lisp	<code>(active compo)</code>
Prolog	<code>ice_Active(+Compo, Result).</code> <code>ice_Active(+Compo).</code>
Tcl	<code>compo active</code>

Parameters

`compo` - The structure/object representing the component

Discussion

The C function returns a pointer to the component now active. All other languages return a boolean value deciding if the component in question is now active.

There are two predicates for prolog. `ice_Active/1` succeeds if the component given as argument is active. `ice_Active/2` succeeds always and returns 1 if the component is active, 0 else.

Example

C	<code>active_compo = Ice_ActiveCompo();</code>
C++	<code>me.Active()</code>
Lisp	<code>(active me)</code>
Prolog	<code>ice_Active(Me).</code>
Tcl	<code>parser active</code>

Errors

None.

7.7 Ice_Activate(): Activate a component

This function is used to switch around between active and inactive components.

Synopsis

```

C          void Ice_Activate( Ice_Compo *compo);
C++       void Component::Activate( void);
Lisp      (activate compo)
Prolog    ice_Activate( +Compo).
Tcl      compo activate

```

Parameters

`compo` – The structure/object representing the component

Discussion

The channels created are local to the currently active component. If you have two components in one single process and want to establish a channel to another process, you have to build two channels, one for each component.

Example

```

C          Ice_Activate( me);
C++       me.Activate()
Lisp      (activate me)
Prolog    ice_Activate( Me).
Tcl      parser activate

```

Errors

None.

7.8 Ice_Detach(): Unregistering

This functions notifies the ILS that the component will stop operation. After calling this function no communication with other components can be done.

Synopsis

```
C          void Ice_Detach( Ice_Compo *compo);
C++       void Component::~~Component( void);
Lisp      (detach compo)
Prolog    ice_Detach( +Compo).
```

```
Tcl      compo detach
```

Parameters

compo - The structure/object representing the component

Discussion

If the component now active is detached, the active component is undefined. You must first activate a component to be further able to communicate.

Example

```
C          Ice_Detach( me);
C++       --
Lisp      (detach me)
Prolog    ice_Detach( Me).
Tcl      parser detach
```

Errors

None.

7.9 Ice_AddCompo(): Creating a base channel

This function establishes basic communication means to another component.

Synopsis

```

C          Ice_Chan *Ice_AddCompo( char *name);
C++       Channel::AddCompo( char *name);
Lisp      chan ←— (addcompo name)
Prolog    ice_AddCompo( +name, -Chan).
Tcl      Ice_Channel chan -to name

```

Parameters

- name** – The name of the component that shall be the communication partner. This has to be the same name as in the `Ice_Attach()`-call of the other component.
- chan** – Return value. A structure representing the internals of the base channel. Normally, no component cares about the content of this.

Discussion

This function builds up a representation of the partner component. It is not guaranteed, however, that such a component already exists. The physical connection is made upon request, i.e., by sending a message to that component or receiving a message from there.

This routine is called automatically if a generic receive results in a message from a component unknown until then. If you later on choose to create a base channel on which you received a message already, no duplicate structure is built, but instead you get the automatically established channel back.

Note that the construction of base channels is local to the currently active component.

Example

```
C          dest = Ice_AddCompo( "semantics");
C++        Channel dest;
           dest:AddCompo( "semantics");
Lisp       (setf dest (addcompo "semantics"))
Prolog     ice_AddCompo( 'semantics', Dest).
Tcl        Ice_Channel dest -to semantics
```

Errors

None.

7.10 Ice_RemoveCompo(): Remove a component

This routine closes the communication links to the other component.

Synopsis

```

C          void Ice_RemoveCompo( Ice_Chan *chan);
C++       void Channel::~~Channel( void);
Lisp      (removecompo chan)
Prolog    ice_RemoveCompo ( +Chan).
Tcl      chan destroy

```

Parameters

`chan` – The structure/object representing the base channel to the other component.

Discussion

The data structures holding the internal values for the data path to the other component are deleted. It is unsafe to attempt communication with a removed component.

Example

```

C          Ice_RemoveCompo( dest);
C++       --
Lisp      (removecompo dest)
Prolog    ice_RemoveCompo( Dest).
Tcl      dest destroy

```

Errors

None.

7.11 Ice_AddChan(): Create an additional channel

This function establishes one additional channel to another component.

Synopsis

C	<code>Ice_Chan *Ice_AddChan(char *name, Ice_Chan *basechan, int flags);</code>
C++	<code>Channel::AddChan(char *name, Channel *basechan, int flags);</code>
Lisp	<code>chan ← (addchan name basechan &key synch raw token)</code>
Prolog	<code>Ice_AddChan(+name, +Basechan, +flaglist, -Chan).</code>
Tcl	<code>Ice_Channel chan -to compo -name name [-synch] [-raw] [-token]</code>

Parameters

name	- The name of the channel to be created.
basechan	- The structure/object that represents the base channel to the component, along which an additional channel shall be created.
compo	- The component that is the remote point of this channel.
flags	- Describes what properties the channel should have. For C and C++ this is an int or'ed together from the values in table 7.3. The call in Lisp uses key parameters that can be set to t. The prolog call uses a list of Flags that stem from table 7.4.
chan	- Return value. A Structure representing the internals of the additional channel. Normally, no component cares about the content of this.

Discussion

This routine is used to create an additional channel to another component. It is required that there exists an base channel to the desired component prior to this call.

Name	Description
ICE_NORMAL	Normal operation like on the base channel
ICE_SYNCH	Synchronous operation. The component waits until the message is delivered before returning from a send request
ICE_RAW	Don't use XDR encoding for messages
ICE_TOKEN	Use a token to control send licensing

Table 7.3: Valid channel flags for C and C++

Name	Description
ice_synch	Synchronous operation. The component waits until the message is delivered before returning from a send request
ice_raw	Don't use XDR encoding for messages
ice_token	Use a token to control send licensing

Table 7.4: Valid channel flags for Prolog

Note that the creation of an additional channel depends on both endpoints of communication. Consequently there is no automatic creation of these channels.

Note also that you may send messages on a special channel if the target components are attached to the system, but that the receiving of that messages depends on the creation of the additional channel by the target components.

Example

```

C          control = Ice_AddChan( "controller", dest,
                    ICE_RAW);
C++       control:AddChan( "controller", dest, ICE_RAW);
Lisp      (setf control (addchan "controller" dest (raw
                    t)))
Prolog    ice_AddChan( 'controller', Dest, [ice_raw],
                    Control).
Tcl       Ice_Channel control -to semantics -name
                    controller -raw

```

Errors

None.

7.12 Ice_RemoveChan(): Stop communicating with a partner

This routine closes the additional channel to the other component.

Synopsis

```

C          void Ice_RemoveChan( Ice_Chan *chan);
C++       void Channel::~~Channel( void);
Lisp      (removechan chan)
Prolog    ice_RemoveChan ( +Chan).
Tcl      chan destroy

```

Parameters

chan - The structure/object representing the additional channel to the other component.

Discussion

The data structures holding the internal values for the data path to the other component are deleted. It is unsafe to attempt communication on a removed channel.

Example

```

C          Ice_RemoveChan( control);
C++       --
Lisp      (removechan control)
Prolog    ice_RemoveChan( Control).
Tcl      control destroy

```

Errors

None.

7.13 Ice_Send(): Send a message

This routine sends a message to another component on a channel.

Synopsis

C	int Ice_Send(Ice_Chan *chan, int msgtag, int turn_Id, char *concern, int t_start, int t_end, int prio, int m_type, void *m_data);
C++	int Channel::send(int msgtag, int turn_Id, char *concern, int t_start, int t_end, int prio, int m_type, void *m_data); int Channel::send(char value); int Channel::send(char *value); int Channel::send(int value); int Channel::send(unsigned int value); int Channel::send(long value); int Channel::send(unsigned long value); int Channel::send(float value); int Channel::send(double value);
Lisp	result ← (send chan msgtag turn-id concern t-start t-end prio m-type m-data) (sendshort chan m-data)
Prolog	ice_Send(-Res, +Chan, +Msgtag, +Turn_Id, +Concern, +T_start +T_end, +Prio, +M_type, +M_data).
Tcl	chan send msgtag turn_Id concern t_start t_end prio m_type m_data

Parameter

- chan** – The medium for message transport. It is a channel that was connected using `Ice_AddCompo()` or `Ice_AddChan()`.
- msgtag** – The message tag that characterizes a message. This can be either `ICE_NOTAG` to send messages with no specified tag or any previously created tag. In case you send a message along an additional channel this must be `ICE_NOTAG`, since no tagged messages are allowed there. The abbreviated calls use `ICE_NOTAG` as value for this parameter.
- turn_Id** – Turn id this message belongs to. The counting of turn ids should start with 1 and increase every time a new turn is processed. Note that ICE does not track the turn ids, the bookkeeping is entirely left to the implementor of components. Note also that in case a function call does not provide a parameter for the turn id, ICE assumes that the turn didn't change since the last time a more elaborate call was done. If such a short call is the first one to be done, an error message is issued. See also `SetTurnId()` for a function to set the actual turn id.
- concern** – You can provide the concern of the message here. The string parameter has no internal semantics, its meaning is left to the implementor and has to be specified together with other interface data.
- t_start** – Start time of this message. We assume that most messages describe hypotheses about certain intervals in time. `t_start` is the point in time at which the description starts. It is measured in 100 ns. If you don't need this values, simply set it to 0. The abbreviated calls use 0 as value for this parameter.
- t_end** – End time of this hypotheses. The abbreviated calls use 0 as value for this parameter.
- prio** – We further assume that most messages describe ranked hypotheses. This ranking ($\in \{0 \dots 100\}$) is given with this parameter. The abbreviated calls use 50 as value for this parameter.
- m_type** – This parameter determines the data type sent as message. See table 7.2 for a list of predefined data types. This set can be extended by use of IDL, the Intarc DDL Layer.
- m_data** – Message data. This is a pointer to the data for C and C++, the data itself is given for Lisp, Prolog and Tcl.

result – The result parameter has only an effect if you chose not to wait for all target components of a message having attached to the application. You may do so by calling `Ice_Configure(NOBLKSEND, 0, "")`. In this case, if one of the target components of a message (it can be more than one due to the configuration of split channels) is not already present, a value of -1 is returned. Otherwise, 1 is returned.

Discussion

Using this function you can send all types of data along a channel to a receiver. For polymorphous languages (C++, Lisp) we provide a shorter function call that reduces typing overhead by assuming default values for several parameters.

Note that the abbreviated calls use the turn id specified by `Ice_SetTurnId()` during their execution.

Example

```
C      res = Ice_Send( dest, ICE_NOTAG, 1, "none",
                    500, 700, 60, IDL_STRING, "NP" );

C++    res = dest.send( ICE_NOTAG, 1, "none", 500,
                    700, 60, IDL_STRING, "NP" );
        res = dest.send( "Hia!" );

Lisp   (setf res (send dest +ICE_NOTAG+ 1 "none" 500
                    700 60 +IDL_STRING+ "NP"))
        (setf res (sendshort dest "Hia!"))

Prolog ice_Send( Res, Dest, ice_notag, 1, 'none',
                    500, 700, 60, idl_string, 'NP' ).

Tcl    dest send ICE_NOTAG 1 "none" 500 700 60
        IDL_STRING "NP"
```

Errors

None.

7.14 Ice_Broadcast(): Send a message to all components

This routine sends a message to all components participating in an ICE system.

Synopsis

C	<code>int Ice_Broadcast(int msgtag, int turn_Id, char *concern, int t_start, int t_end, int prio, int m_type, void *m_data);</code>
C++	<code>int Channel::Broadcast(int msgtag, int turn_Id, char *concern, int t_start, int t_end, int prio, int m_type, void *m_data);</code>
Lisp	<code>res ← (broadcast chan msgtag turn-id concern t-start t-end prio m-type m-data)</code>
Prolog	<code>ice_Broadcast(-Res, +Msgtag, +Turn_Id, +Concern, +T_start +T_end, +Prio, +M_type, +M_data).</code>
Tcl	<code>chan broadcast msgtag turn_Id concern t_start t_end prio m_type m_data</code>

Parameter

- `chan` – This parameter is only present in the object-oriented implementations (for C++, Lisp and Tcl). There is no functional reason for this; the usage may vanish in future releases of ICE.
- `msgtag` – The message tag that characterizes a message. This can be either `ICE_NOTAG` to send messages with no specified tag or any previously created tag.
- `turn_Id` – Turn id this message belongs to. The counting of turn ids should start with 1 and increase every time a new turn is processed. Note that ICE does not track the turn ids, the bookkeeping is entirely left to the implementor of components. Note also that in case a function call does not provide a parameter for the turn id, ICE assumes that the turn didn't change since the last time a more elaborate call was done. If such a short call is the first one to be done, an error message is issued. See also `SetTurnId()` for a function to set the actual turn id.
- `concern` – You can provide the concern of the message here. The string parameter has no internal semantics, its meaning is left to the implementor and has to be specified together with other interface data.
- `t_start` – Start time of this message. We assume that most messages describe hypotheses about certain intervals in time. `t_start` is the point in time at which the description starts. It is measured in 100 ns. If you don't need this values, simply set it to 0. The abbreviated calls use 0 as value for this parameter.
- `t_end` – End time of this hypotheses. The abbreviated calls use 0 as value for this parameter.
- `prio` – We further assume that most messages describe ranked hypotheses. This ranking ($\in \{0 \dots 100\}$) is given with this parameter. The abbreviated calls use 50 as value for this parameter.
- `m_type` – This parameter determines the data type sent as message. See table 7.2 for a list of predefined data types. This set can be extended by use of IDL, the Intarc DDL Layer.
- `m_data` – Message data. This is a pointer to the data for C and C++, the data itself is given for Lisp, Prolog and Tcl.

result – The result parameter has only an effect if you chose not to wait for all target components of a message having attached to the application. You may do so by calling `Ice_Configure(NOBLKSEND, 0,)`. In this case, if one of the target components of a message (it can be more than one due to the configuration of split channels) is not already present, a value of `-1` is returned. Otherwise, `1` is returned.

Discussion

This function is used to send messages to all components participating in an distributed application using ICE. These components need not be known to the component issuing the broadcast call.

Example

```

C          res = Ice_Broadcast( ICE_NOTAG, 1, "none",
                    500, 700, 60, IDL_STRING, "NP");
C++       res = dest.broadcast( ICE_NOTAG, 1, "none",
                    500, 700, 60, IDL_STRING, "NP");
Lisp      (setf res (broadcast dest +ICE_NOTAG+ 1 "none"
                    500 700 60 +IDL_STRING+ "NP"))
Prolog    iceBroadcast( Res, Dest, ice_notag, 1,
                    'none', 500, 700, 60, idl_string, 'NP').
Tcl      dest broadcast ICE_NOTAG 1 "none" 500 700 60
                    IDL_STRING "NP"

```

Errors

None.

7.15 Ice_GetSentModuleId(): Retrieve module Id

This routine retrieves the module id of a message just sent.

Synopsis

```
C      char *Ice_GetSentModuleId( Ice_Compo
      *component);
C++    char *Component::GetSentModuleId ( void);
Lisp   string ← (get-sent-module-id component)
Prolog ice_GetSentModuleId( +Component -Module_Id).
Tcl    module_Id ← component getSentModuleId
```

Parameter

- component - This parameter specifies the component for which the module id of the last message should be retrieved.
- module_Id - This return value contains the module id of the last message sent.

Discussion

Example

```
C      lastId = Ice_GetSentModuleId( me);
C++    lastId = me.GetSentModuleId();
Lisp   (setf last-id (get-sent-module-id me))
Prolog ice_GetSentModuleId( Me, LastId).
Tcl    set lastId [me getSentModuleId]
```

Errors

None.

7.16 Ice_SetTurnId(): Set turn id for following messages

This routine is used to set the turn id to be used by ICE in the messages to come where no turn id is supplied.

Synopsis

```

C          ---
C++       void Component::SetTurnId ( int turn_Id);
Lisp      (set-turn-id component turn-id)
Prolog    ---
Tcl       ---

```

Parameter

`component` - This parameter specifies the component for which the module id of the last message should be retrieved.

`turn_Id` - This parameter specifies the turn id to be used from now on.

Discussion

The provided turn id is valid until another call to `SetTurnId()` is issued or a sending call is used that supplies a turn id.

Note that sending calls which provide a turn id don not change the turn id set with this function. It is preserved for further abbreviated sending calls.

Example

```

C          ---
C++       me.SetTurnId( 2);
Lisp      (set-turn-id me 2)
Prolog    ---
Tcl       ---

```

Errors

None.

7.17 Ice_Receive(): Get a message from other components

This functions blocks until a message with the desired specifications arrives. The message is returned. This function can only be used without the event-oriented version of ICE (See section 7.24 for details).

Synopsis

```

C          int Ice_Receive( Ice_Chan *chan, int msgtag,
                        Ice_Msg *msg);
C++       int Channel::Receive( int msgtag, Ice_Msg
                        *msg);
          int Component::Receive( int msgtag, Ice_Msg
                        *msg);
Lisp      msg, acttag ← (receive chan msgtag)
Prolog    ice_receive( +Chan, +Msgtag, -Acttag, -Msg).
Tcl      msg ← chan receive [msgtag]
         msg ← compo receive [msgtag]

```

Parameter

- chan – This parameter specifies on which channel the function will listen for incoming messages.
- msgtag – You may specify a dedicated message tag to be listened for. Note that since additional channels can't be tagged, this parameter always has to be ICE_NOTAG for them.
- msg – This return value contains a message structure upon returning from the function. The various fields can be retrieved using functions described below.
- acttag – This return value contains the actual received message tag in case that an under-specified tag was given as parameter.

Discussion

This function waits indefinitely. If a component wants to continue processing in case that no message is present, `Ice_Probe()` should be used.

There is no function that discriminates different Data types.

You may under-specify the origin of messages at several levels:

- Get a message on a definite channel and with a definite tag. Note that additional channels are not tagged, thus you have to give `ICE_NOTAG` as message tag. The tag `ICE_NOTAG` serves as standard tag on the base channel, so it is the default value there.
- Get a message on a definite channel disregarding the message tag. Supply `ICE_ANY` as message tag. Note that for additional channels this is equivalent to the previous method.
- Get a message from a definite component, disregarding the channel and the message tag. Specify `ICE_CMPTAG` as message tag and use the base channel as receiving channel.
- Get a message from anywhere. Use `ICE_ANY` as message tag and specify `ICE_ANYCMP` as channel. Note that for C++, Tcl and Lisp there is a special method for Components that does exactly this.

Example

```
C      acttag = Ice_Receive( dest, ICE_ANY, &msg);
C++   acttag = dest.Receive( ICE_ANY, &msg);
Lisp   (setf msg (receive dest +ICE_ANY+))
       (multiple-value-bind (msg acttag) (receive
         dest +ICE_ANY+) (write acttag))
Prolog ice_Receive( Dest, ice_any, Acttag, Msg).
Tcl    set msg [dest receive]
```

Errors

None.

7.18 Ice_TReceive(): Receiving with timeout

This functions blocks until a message with the desired specifications arrives. The message is returned. In case no message arrives within the specified time, an error indicator is returned.

Synopsis

```

C          int Ice_TReceive( Ice_Chan *chan, int msgtag,
                        Ice_Msg *msg, long timeout);
C++       int Channel::Receive( int msgtag, Ice_Msg
                        *msg, long timeout);
          int Component::Receive( int msgtag, Ice_Msg
                        *msg, long timeout);
Lisp      msg, acttag ← (receive chan msgtag :timeout
                        timeout)
Prolog    ice_receive( +Chan, +Msgtag, +Timeout,
                        -Acttag, -Msg).
Tcl       msg ← chan treceive timeout [msgtag]
          msg ← compo treceive timeout [msgtag]

```

Parameter

- chan – This parameter specifies on which channel the function will listen for incoming messages.
- msgtag – You may specify a dedicated message tag to be listened for. Note that since additional channels can't be tagged, this parameter always has to be `ICE_NOTAG` for them.
- msg – This return value contains a message structure upon returning from the function. The various fields can be retrieved using functions described below.
- timeout – This parameter specifies the amount of time in Milliseconds that ICE should wait for a message before returning unsuccessfully.
- acttag – This return value contains the actual received message tag in case that an under-specified tag was given as parameter. It also specifies if the receiving was succesful within the specified time amount. If not, it contains 0.

Discussion

None.

Example

```
C      acttag = Ice_TReceive( dest, ICE_ANY, &msg,
                          500);
C++    acttag = dest.Receive( ICE_ANY, &msg, 500);
Lisp   (setf msg (receive dest +ICE_ANY+ :timeout
                    500))
        (multiple-value-bind (msg acttag) (receive
                    dest +ICE_ANY+ :timeout 500) (write acttag))
Prolog  ice_Receive( Dest, ice_any, 500, Acttag, Msg).
Tcl     set msg [dest receive 500]
```

Errors

None.

7.19 Ice_Probe(): Check for incoming messages

This function checks if any pending messages are present. This function can only be used without the event-oriented version of ICE (See section 7.24 for details).

Synopsis

```

C          int Ice_Probe( Chan *chan, int msgtag);
C++       int Channel::Probe( int msgtag);
          int Component::Probe( int msgtag);
Lisp      result ← (probe chan msgtag)
Prolog    ice_probe( +Chan, +Msgtag, -Result).
          ice_probe( +Chan, +Msgtag).
Tcl       result ← chan probe [msgtag]
          result ← compo probe [msgtag]
```

Parameters

- `chan` - This parameter specifies which channel the function will probe for incoming messages.
- `msgtag` - You may specify a dedicated message tag to be checked for. Note that since additional channels can't be tagged, this parameter always has to be `ICE_NOTAG` for them.
- `result` - This return value is 0 if no messages can be received. It would be 1, if there were any messages that could be received.

Discussion

The semantics of the parameters is the same as for the `Ice_Receive()`-call. Note that no message is actually received. If `Ice_Probe()` yields 1, you have to receive a message explicitly.

There are two predicates for Prolog. `ice_probe/2` succeeds if there is a pending message, otherwise it fails. `ice_probe/3` succeeds always and returns 1 if a message is pending, 0 else.

Example

```
C      acttag = Ice_Probe( dest, ICE_ANY);
C++    acttag = dest.Probe( ICE_ANY);
Lisp   (setf msg-there (probe dest ICE_ANY))
Prolog ice_Probe( Dest, ice_any, YesNo).
        ice_Probe( Dest, ice_any).
Tcl    set msg_there [semantics probe]
```

Errors

None.

7.20 Ice_SetMinTurnId(): Set the minimum turn id for following receives

This routine is used to set the minimum turn id. Only messages with turn ids greater or equal to the one specified are delivered to a component.

Synopsis

C	<code>void Ice_SetMinTurnId(Ice_Compo *compo, int turn_Id);</code>
C++	<code>void Component::SetMinTurnId (int turn_Id);</code>
Lisp	<code>(set-min-turn-id component turn-id)</code>
Prolog	<code>ice_SetMinTurnId(+Compo +Turn_Id).</code>
Tcl	<code>compo setMinTurnId turn_Id</code>

Parameter

<code>compo</code>	-	This parameter specifies the component for which the minimum turn id should be set.
<code>turn_Id</code>	-	This parameter specifies the turn id to be used as lower bound from now on.

Discussion

The turn id is stored as data within messages. That way, a call to `Ice_Probe()` may succeed while the following `Ice_Receive()` infinitely waits in case that the only message present belongs to an earlier turn.

Example

C	<code>Ice_SetMinTurnId(me, 2);</code>
C++	<code>me.SetMinTurnId(2);</code>
Lisp	<code>(set-min-turn-id me 2)</code>
Prolog	<code>ice_SetMinTurnId(Me, 2).</code>
Tcl	<code>me setMinTurnId 2</code>

Errors

None.

7.21 Access functions for message data

Several functions retrieve information out of an message structure.

Synopsis

```
Lisp      result ← (get-t-start msg)
          result ← (get-t-end msg)
          result ← (get-t-prio msg)
          result ← (get-m-type msg)
          result ← (get-turn-id msg)
          result ← (get-module-id msg)
          result ← (get-concern msg)
          result ← (get-m-data msg)
          result ← (get-sender msg)
```

Parameters

`msg` - This should be a message constructed by the ICE receive functions.

`result` - This return value contains the desired field of the message upon return.

Discussion

In C and C++ a message is a struct with the following definition:

```
typedef struct Ice_Msg {
    Ice_Chan *channel;      /* Channel to transport this message */
    Ice_Compo *sender;     /* Which component sent this message */
    int t_Start;           /* Time slice start value */
    int t_End;             /* Time slice end value */
    int prio;              /* Priority (or rating) */
    int m_Type;           /* Data type */

    unsigned int turn_Id; /* Turn the message belongs to */

    char *module_Id;      /* Unique id of this message */
    char *concern;        /* The concern of this message */
    void * m_Data;        /* Pointer to data */
} Ice_Msg;
```

You can easily get data out of it. The definitions for Lisp follow from the synopsis, they simulate accessors to embedded data of a `msg`-object. The access to message field for Prolog is easy, too. The receiving functions return a compound term with functor `ice_msg` and arity 6. The parameters of the term denote the sender, start time, end time, priority, message data type and message data, respectively, so it looks like this:

```
ice_msg( Sender, T_Start, T_End, Prio, M_Type,
        Turn_Id, Module_Id, Concern, M_Data)
```

To store message information in Tcl, an array `Ice_Msg` exists that holds the appropriate data. Note that the array is updated every time a message is received, it is not conserved. The array elements are:

Element name	Content
<code>Ice_Msg(acttag)</code>	The actual message tag that was received
<code>Ice_Msg(sender)</code>	The component which sent this message
<code>Ice_Msg(t_start)</code>	The start time of the described interval
<code>Ice_Msg(t_end)</code>	The end time
<code>Ice_Msg(prio)</code>	The priority of the data
<code>Ice_Msg(m_type)</code>	The message data type
<code>Ice_Msg(turn_Id)</code>	The turn id for this message
<code>Ice_Msg(module_Id)</code>	The unique id of this message
<code>Ice_Msg(concern)</code>	The concern

Example

```
Lisp      (setf starttime (get-t-start msg))
```

Errors

None.

7.22 Idl_UserInit(): Initialization of IDL

This function initializes IDL.

Synopsis

C	<code>void Idl_UserInit(void);</code>
C++	<code>void Idl_UserInit(void);</code>
Lisp	<code>(idl-userinit)</code>
Prolog	<code>idl_UserInit.</code>
Tcl	<code>idl_UserInit</code>

Parameters

None.

Discussion

This function replaces `Ice_Init()` if you specified user-defined complex data structures.

Examples

C	<code>Idl_UserInit();</code>
C++	<code>Idl_UserInit();</code>
Lisp	<code>(idl-userinit)</code>
Prolog	<code>idl_UserInit.</code>
Tcl	<code>Idl_UserInit</code>

Errors

None.

7.23 Idl_Free(): Deallocate memory

This function releases memory allocated by IDL-Routines.

Synopsis

```
C          void Idl_Free( int m_type, void *m_data);  
C++       void Idl_Free( int m_type, void *m_data);  
Lisp      --  
Prolog    --
```

Parameter

`m_type` - This specifies the type of the data element to be deallocated.
`m_data` - A Pointer to the data to be deallocated.

Discussion

In a further release the detection of the actual data type will be handled automatically.

Examples

```
C          Idl_Free( IDL_STRING, s);  
C++       Idl_Free( IDL_STRING, s);  
Lisp      --  
Prolog    --
```

Errors

None.

7.24 Ice_Handler(): Event-based message handler

This functions sets up a concurrent message handler that receives messages asynchronously and provides the program with messages.

Synopsis

```

C          ---
C++       ---
Lisp      ---
Prolog    ---
Tcl       Ice_Handler name procedure

```

Parameters

- name** - This parameter specifies the name of the message handler object. It is introduced into the Tcl-hierarchy of names so the name should start with a dot (.).
- procedure** - This parameter names the Tcl procedure to be called if a message arrives.

Discussion

This function is only available if ICE was compiled using the event-oriented schema (see the file `#{ICE_ROOT}/Definitions` about how to do that). Currently, it is only implemented for Tcl and only on SPARCstations using Solaris 2.3/2.4.

The idea behind this extension is not to be forced to execute a `Ice_Receive()` if one wants to get a message. Since this function blocks, no Tcl-application would ever react upon user interactions such as button-clicking etc.. `Ice_Probe()` on the other hand, which would enable to continue with the user-interaction, produces overhead by busy-waiting.

So we designed an event-based extension that uses Solaris-Threads to concurrently wait for messages via `Ice_Receive()`. If a message is received, an X-event is generated and sent to the main Tcl-application (more or less comparable to the `send`-command in Tcl). In response to the event, the Tcl interpreter chooses to execute a event-handler which in turn calls a user-definable Tcl procedure. That way, messages get processed immediately without disturbing normal operation.

There are currently some things to note is you use the event-based schame within Tcl-programs:

- Only one component can be attached.
- No `Ice_Receive()`-calls are allowed if the handler is currently active (see below).

Example

```
C          ---
C++        ---
Lisp       ---
Prolog     ---
Tcl        Ice_Handler .ice sender_handler
```

Errors

None. Both threads in Tcl-programs are synchronized by semaphores that should cover 95% of the deadlock and race conditions. If, however, you detect some strange behavior, send us examples of what went wrong...

7.25 Ice_Handler(): Event-based message handler: Widget commands

This section describes the command valid for an `Ice_handler`-object .

Synopsis

```

C          ---
C++       ---
Lisp      ---
Prolog    ---
Tcl       name activate
            name deactivate
            name proc procedure
            msg ← name get

```

Parameters

- `name` – This parameter specifies the name of the message handler object. It has to be already introduced into the Tcl-hierarchy of names via a call to the Tcl command `Ice_Handler`.
- `procedure` – This parameter names the Tcl procedure to be called if a message arrives.

Discussion

These functions are only available if ICE was compiled using the event-oriented schema (see the file `${ICE_ROOT}/Definitions` about how to do that). Currently, it is only implemented for Tcl and only on SPARCstations using Solaris 2.3/2.4.

The effects of the widget commands are:

- **activate**: The event handler is activated. This results in a continuous call to `Ice_Receive()` to get messages.
- **deactivate**: The event handler is deactivated. No more calls to `Ice_Receive()` are issued. note that there still may be one pending message to be processed if the deactivation was requested by the message handling Tcl-procedure.
- **proc**: Specifies that from now on a different procedure should be used to handle incoming messages. This change is effective immediately or start-

ing from the next message to be handled if issued from within the actual message ‘-handling procedure.

- **get**: Using this method a pending message is delivered to the Tcl program. The method returns a message structure just like the **receive** method does. It is an error to call this method outside of the message-handling procedure.

To get an impression how the methods work, you should consider to consult the demo program `${ICE_ROOT}/demos/tcl-event-demo`.

Example

```
C          ---
C++       ---
Lisp      ---
Prolog    ---
Tcl       .ice activate
          .ice deactivate
          .ice proc other_handler
          set msg [.ice get]
```

Errors

None.

References

- Amtrup, Jan W. 1994. ICE-Intarc Communication Environment: Design und Spezifikation. Verbmobil Memo 48, Universität Hamburg, Hamburg, September.
- Burns, Alan. 1988. *Programming in OCCAM 2*. Reading, Ma.: Addison-Wesley.
- Corbin, John R. 1990. *The Art of Distributed Applications*. Sun Technical Reference Library. New York: Springer-Verlag.
- Geist, Al, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. 1994. Pvm3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Te., May.
- Graham, Ian and Tim King. 1990. *The Transputer Handbook*. London et. al.: Prentice Hall.
- Hoare, Charles A. Richard. 1978. communicating Sequential Processes. *Communications of the ACM*, 21(8):666-677, August.
- Pyka, Claudius. 1992. Schnittstellendefinition mit ASL-DDL. ASL-TR 42-92/UHH, Universität Hamburg, March.