CARAVEL

API for Caravel OS/400 Framework

BASE100

BASE 100, S.A.
www.base100.com

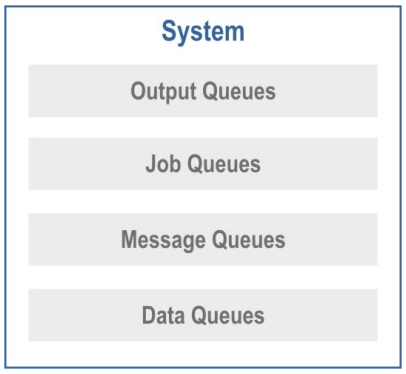
Document: api_caravel_os400_framework_v1a_en.doc
Version: 1.0

BASE 100, S.A.
http://www.base100.com

Contents

Chapter 1. Introduction

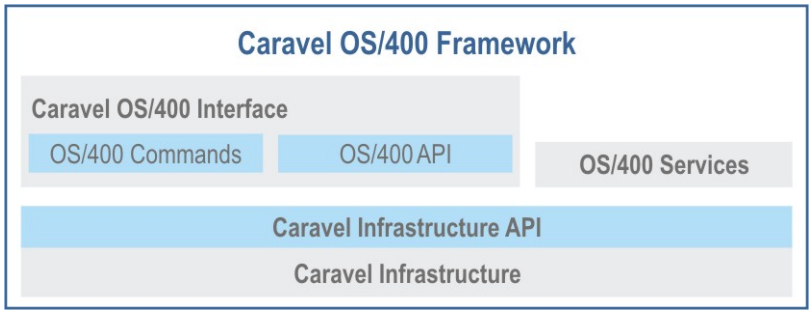
Caravel OS/400 Framework is made up of a set of classes and interfaces where the system services and characteristics offered by an OS/400 are implemented and managed and on which the whole Caravel is based.



The system offers the following services:

- Out queues management service (lists).
- Management service for the Batch process (submitted jobs).
- Message management service (queues and files).
- Data interchange management service.

This generates support to control the different resources generated by an application and sent to the system, such as reports, messages or data, by means of a group of associated queues with a determined service.



Caravel OS/400 Framework subdivides into:

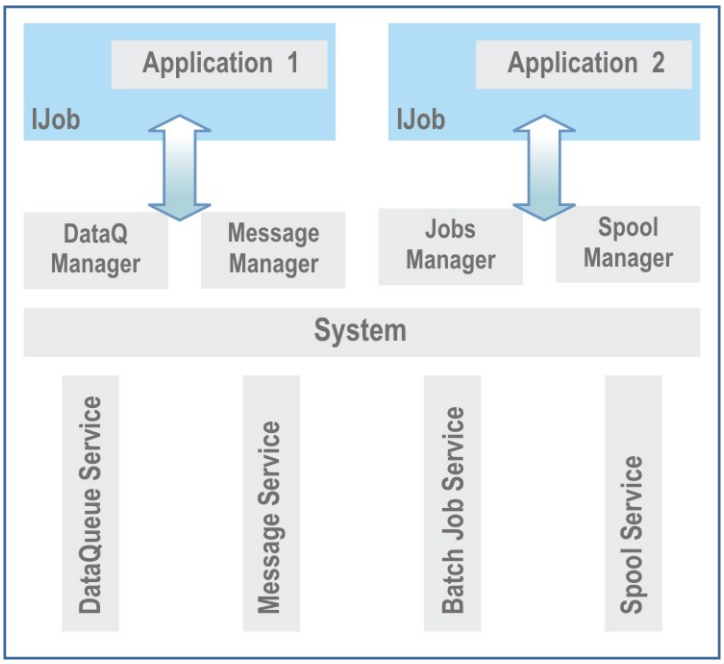
- *Caravel infrastructure.* This is the name given to the set of classes and interfaces that implement the system in itself. Its public part is *Caravel Infrastructure API*. It is a set of interfaces that the user can accede from his own classes to know and modify the system contents and state, by means of the manager for the different services offered by the system.
- *Caravel 0S/400 Interface.* It provides a system operation interface through a group of commands *(OS/400 Commands)* for the management of the different services offered by the system and an API *(OS/400 API).*
- For its implementation the *Caravel infrastructure API* has been used, this fact makes it easily extensible, since new commands can be added to the group offered by *OS/400 Commands*.
- *OS/400 Services are* processes that manage system information with a determined aim. For example, the spool of out queue lists sends to the corresponding printer the lists that are ready; or the process executor launches the execution of the necessary out queue processes.

By using the *Caravel Infrastructure API* functions, it is possible to send information from the programs of an application to certain services offered by the system, such as the spool service (out queues). It is also possible to develop a consultation interface and a modification of the information operated by the services, extending in this way the *Caravel OS/400 Interface*.

This document presents the C*aravel Infrastructure API* set of classes that allows the Caravel to offer a series of uses and elements as the ones offered by an OS/400 system.

Chapter 2. Caravel system

A Caravel application in execution will be connected to only one system, which can provide its spool management services, data queues, etc. to different applications.



The execution process of a Caravel application is represented by an object of the *Ijob class* (in *com.transtools.caravel.job*) and it is through this object and by means of the methods it provides that the *managers* that will give access to the system services to which the application is connected can be obtained. A service manager can accede all the elements operated by the service; each element is called *entity.*

This is how the spool service supplies a manager that can accede all the data out queues in the service; each of them is an entity.

The entities in the different system services need a physical implementation where the necessary information can be stored in a persistent way. The physical support can be:

- A relational database called *entity database.* It is the option currently implemented. For further details about its structure, consult the *Entity DB Structure document*.

The physical implementation of the system entities does not condition the way of obtaining or modifying its information.

To obtain the *Ijob* that represents the execution process, the *getJob ()* method is used and it provides the base classes of language emulation CL, RPG and ILE-RPG. Through this object, the different system service *managers* will be obtained (data queues, messages, jobs, etc.) which will allow operating the service entities.

```
IDataQueueManager queueManager = getJob().getDataQueueManager();
```

In this example, the object *queueManager* operates all the entities that the system has in its data queue service.

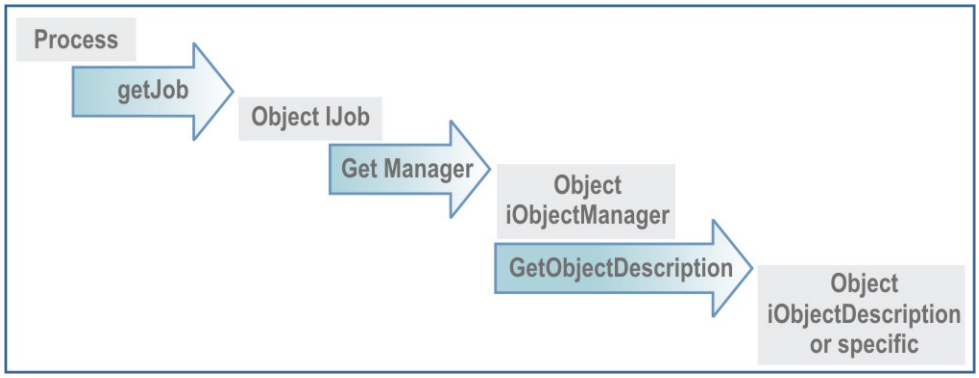
All the service *managers* derive from the *IobjectManager* class and they have the necessary functionality to add, eliminate and modify entities (if proceeding) in the service they run. They also have a series of methods that allow obtaining a determined entity or going through the ones, the service has, by means of an *iterator (object Iterator class).*

The interface that represents an entity is the *IObjectDescription*, defined in the package *com.transtools.caravel.system*, although some services have more specialized interfaces to define the entities they manage.

```
Iterator it = queueManager.getAllObjectsDescriptions();
while (it.hasNext()){
   IobjectDescription objectDesc = (IObjectDescription) it.next();
   System.out.println("Object named "+objectDesc.getName()
         +" belongs to library "+objectDesc.getLibrary());
}
```

In this example an *iterator* about all the entities stored by the data queue service is obtained, managed by the object queueManager.

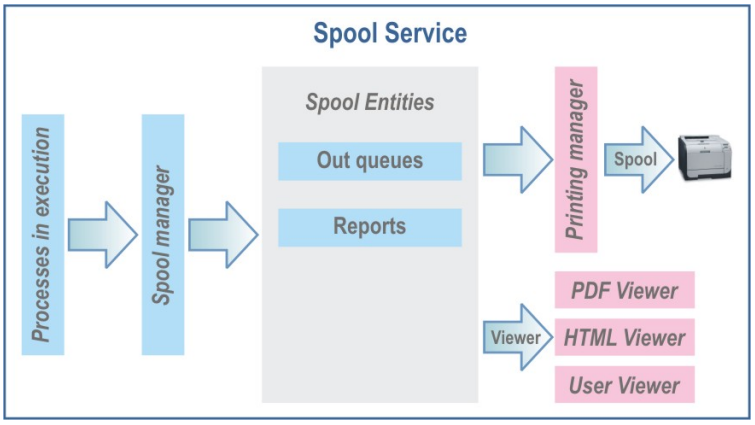
The following graph summarizes how to obtain entities from a determined service of the execution process:



Chapter 3. Spool service

The spool service allows placing the lists in different printing queues to execute lately any kind of operation (hold, print, see, etc.).

Various types of interrelated entities such as the out queues, the printer definition and the data sent to the queues take part in this service management.



The execution processes will send the generated data to the spool service out queues. The spooler, process that belongs to the *OS/400 Services*, will consult the service entities to see which data should be printed, sending the ones that are ready to the printer specified in the data characteristics.

The interface that represents this system manager is *IOutQueueManager*, from the package *com.transtools.caravel.system.outq*.

To obtain a spool service *manager* used by the execution process the method *getOutQueueManager()* of the interface *Ijob will be used.*

```
IOutQueueManager outManager = getJob().getOutQueueManager();
```

Apart from this class, for the list management the interfaces defined in *com.transtools.caravel.report* can be used, they provide access to the queue lists. They are, among others:

- *IOutQueue,* represents an out queue type entity and it has methods to access its own characteristics and data.
- *IReport*, represents a report queued in an out queue entity.

Once the spool service is completed, the interface *IPrintingManager*, defined in *com.transtools.caravel.print*, provides the functionality for the report printing and represents a service used by the spool service.

Through the manager, a determined queue can be obtained with *findOutQueue()*.

```
IOutQueue outQueue = outManager.findOutQueue("LIBRARY", "NAME");
```

Once the queue is obtained, it is possible to have access to the reports it contains, they are objects of the *Ireport* class, either by looking for a determined report through *findReport()* or by getting an iterator about the reports it contains with *getReportsIterator()*.

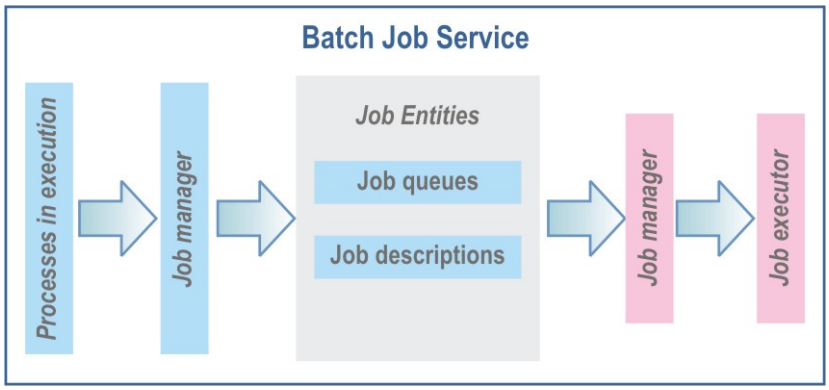
```
IPrintingManager prtMan = getJob().getPrintingManager();

Iterator it = outQueue.getReportsIterator();
while (it.hasNext()) {
  IReport report = it.next();
  if (report.isReady())
    prtMan.printReport(report);
}
```

Chapter 4. Batch process service

A command or program execution can be deferred or done in batch mode, this means that it is possible to launch its execution ordering it not to be immediate but to occur at a certain moment or to be activated by an operator.

The batch or submitted jobs process service manager with its functionality provides the way to arrange the jobs executed in such a way that it is possible to decide when they are going to be executed, to modify the priority over other processes of the queue, etc.



The processes in execution can invoke others in such a way that their execution should pass through a process queue in which it is stored waiting for the execution order or that an established condition, when it is sent to the queue, is fulfilled, as it can be that it is executed at a determined time or that a third process is in charge of changing its state to allow its execution.

These interfaces intervene in the management of the process queues:

- *IjobDescriptionManager*: It manages the entities that store the jobs descriptions, that is to say, the characteristics of all the processes executed using such description. It is defined in the *com.transtools.caravel.system.jobd* package.
- *IjobQueueManager*: It manages the entities of the process queues type and it is defined in the *com.transtools.caravel.system.jobq package*.
- *IjobInfo*: It provides information about an active job. It is defined in the *com.transtools.caravel.system.job* package.

In this way, the *manager* of the entities of the process queues type of the batch process service can be obtained through the *getJobQueueManager()* method of the *Ijob* interface.

```
IJobQueueManager jobQManager = getJob().getJobQueueManager();
```

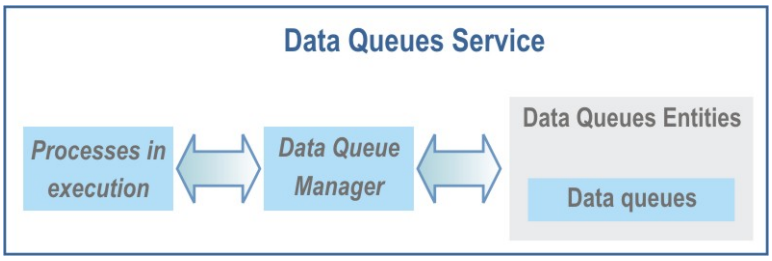
By means of this object and the methods it provides it will be possible to access the different entities it contains.

If you want to have access to the job descriptions of the service the *getJobDescriptionManager()* method will be used to obtain its manager.

```
IJobDescriptionManager jdMan = getJob().getJobDescriptionManager();
```

Chapter 5. Data queues service

The data queues allow the passing of the information among programs, as it gives them a static area of interchange.



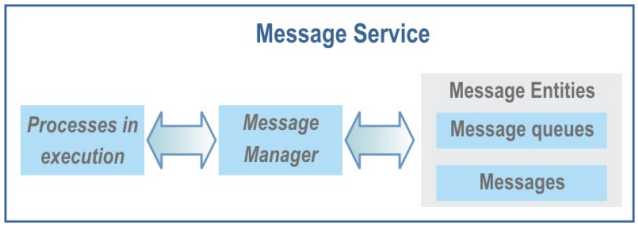
The data queues service allows defining the queues with different characteristics, such as the data type that stores or its capacity, in such a way that the processes will be capable of sharing information sending it to the queue that is necessary.

The interface that provides the functionality for the management of the data queues is *IDataQueueManager*, of the *com.transtools.caravel.system.dtaq* package.

To obtain the system data queues *manager* used by the process in execution, the method *getDataQueueManager()* of the *Ijob* interface will be used.

```
IDataQueueManager queueManager = getJob().getDataQueueManager();
```

Chapter 6. Message service



In an OS/400 system, the messages provide a specialized communication among the programs of an application, consequently *Caravel* implements this characteristic.

The application processes in execution by means of the functionality provided by *Caravel Infrastructure API*, will be capable of sending messages to the entities maintained by the service (message queues). This system provides a mechanism of communication among programs and even with the users, since they will have access to the message.

For example, in a nighttime execution of a series of processes, these can send messages to the properties queue of the system administrator and where he is informed of the state of the termination processes. The administrator user will be able to consult the stored messages in the queue and to eliminate those not needed by other processes.

You can also control the execution of certain processes conditioning them to the existence or not of a concrete message in a system queue.

Several interfaces are provided for the message queues management, all of them defined in the *com.transtools.caravel.system.msgq* package:

- *ImessageQueueManager*. It represents the manager for all the message queues.
- *ImessageQueue.* It represents a message queue entity; this class provides more functionality than the *IObjectDescription*.
- *Imessage.* It represents a message sent or received from a determined message queue.

As it can be seen, more specialized interfaces are used in this management.

To obtain the queue manager the *getMessageQueueManager()* method is used:

```
IMessageQueueManager msgqMan = getJob().getMessageQueueManager();
```

This manager provides several methods that allow obtaining only one of the managing queues, returning an object of the *ImessageQueue* interface.

```
IMessageQueue msgQueue = msgqMan.getMessageQueue("NAME", "LIBRARY");
```

Having an object that represents a queue, you can have access to the messages it stores or send new messages, which will be objects of the *Imessage* class.

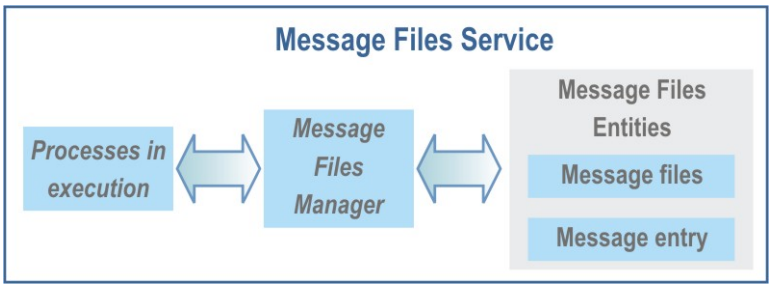
```
Iterator it = msgQueue.getMessageIterator();
while (it.hasNext()){
  IMessage message = (IMessage) it.next();
  System.out.println(message.getMsgType()+" - "+
                        message.getMsgText());
}
```

When doing the casting about the object returned by the iterator, either the class *IMessage* or the *IobjectDescription* can be indicated.

Chapter 7. Message files service

The message files allow centralizing literals and messages frequently used, allowing a greater maintenance facility by limiting the modification of any message to changing its entry in the file where it belongs.

The message files do not belong to the message service, but in a message queue instead of storing any literal you can save a message description stored in a message file.



Instead of using static literals defined in a program, a specified message file can be acceded to obtain that literal, which could be sent to a message queue, to a printer report or be shown on screen.

The *Caravel Infrastructure API* provides several interfaces for the management of message files service. They are defined in the *com.transtools.caravel.system.msgf* package.

- *IMessageFileManager,* represents the manager of all the system message files.
- *IMessageFile* represents an entity of the message files type.
- *IMessageEntry* represents a message file entry.

In order to obtain the message file manager, *IJob* provides the *getMessageFileManager()* method:

```
IMessageFileManager msgFileMan = getJob().getMessageFileManager();
```

Through this object, a specified system file can be accessed.

```
IMessageFile msgFile = msgFileMan.getMessageFile("LIBRARY", "FILE");
```

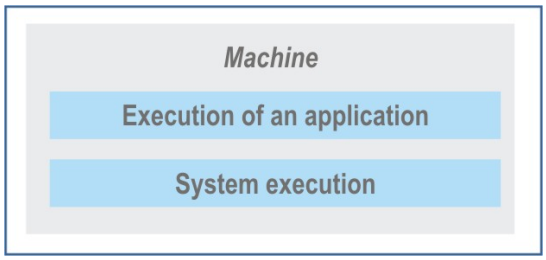
From the object that represents a specified file, it is possible to have access to its contents, that is to say, to the file entries, by the *getMessageFileEntry ()* method, indicating the key to the required entry:

```
IMessageEntry msgEntry = msgFile.getMessageFileEntry("AAA0001");
```

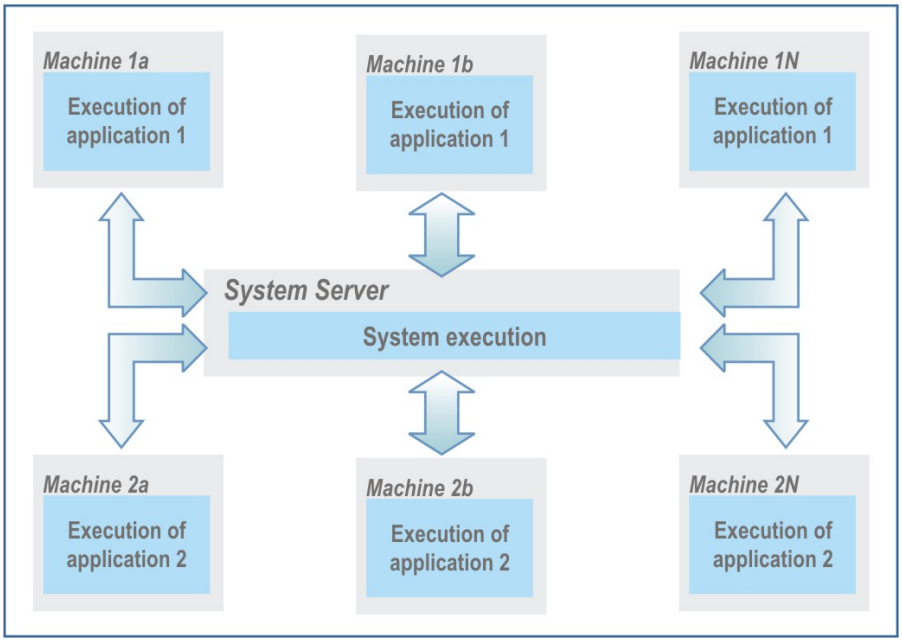
Chapter 8. System Installation Architecture

The *Caravel* system is based on the existence of a physical implementation, this means, a structure where the system entities will be stored and which should be available for the applications required by the system. By means of the API functions, you have access to its contents, independently of the location it is really situated in and of the physical implementation, it has.

- System installed in Local. The functions of access to the system are executed in the same machine where the application is executed and only that post has access to the system information. Although this configuration is possible, it does not allow other job posts to have access to the system information. This type of installation is usual in development and tests.



- System installed in Remote. The application is executed in several machines that will have access to the system whenever needed, but this is installed in a server. The access functionality to the system is executed in the server by RMI.



In any case, the structure where the system is implemented must not necessarily be in the same machine where the access functions are executed.

Chapter 9. OS/400 Services

This part of the *Caravel OS/400 Framework* provides a series of services that are obtained when certain classes are executed.

- System service process.
- Printing spooler process.

System service process

For an application to have access to the system, it must be active and listening to the petitions of the application that uses it, for this there must be a process in execution.

If the access mode to the system is local, this is executed at the same time as the application.

If the access mode is remote, it is necessary to execute a process in the machine that acts as the System Server, and that allows the system to receive the petitions from the applications that are being executed in the client machines by means of RMI. In this case the system service process must be activated, and for this the:

com.transtools.caravel.tools.systemserver.SystemServerRunner

Class must be activated, and it listens to the access petitions to the systems that want to use the applications that connect with that process.

The clients can execute the application installed in them and have direct access to the system or the application can be installed in an applications server and use think clients that send their petitions through such server.

Printing spooler process

This process has been developed having access to the necessary managers and entities through *Caravel Infrastructure API,* its source code is available and can be modified. The executing class is:

com.transtools.caravel.tools.spooler.Spooler

Whenever a list is generated two actions are realized:

- To save the list information, data to be presented and its format in a specified file, located in a directory that corresponds itself with the out queue chosen to send the list.
- To register the report in the entity that represents the out queue, assigning it a state.

Although the report state indicates that it is ready to be printed, this action will not be executed until the order is sent. This is the *spooler* process that is in charge of reading the entity information and verifying if there is any report ready to be printed, sending it to the printer assigned to its out queue.

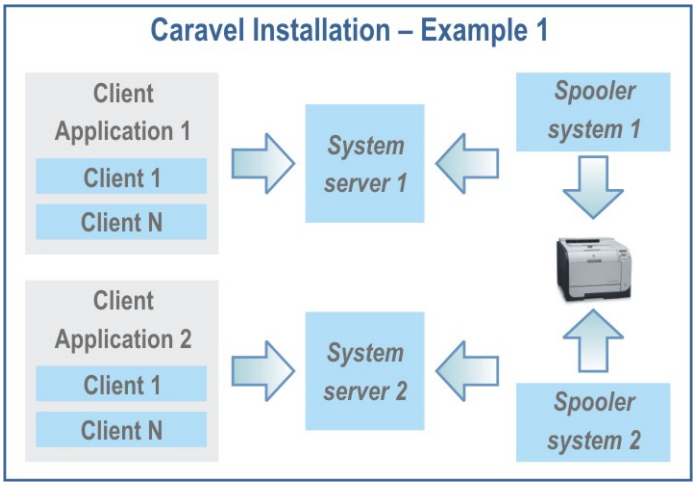
The *spooler* printing process can revise the lists of all the printing queues or the lists of the queues associated to a specified set of printers, so you can have as many *spooler* processes as necessary. In any case, this process should be executed in a machine that allows it to recognize all the printers to which it will send the reports.

Being the *spooler* a process that has access to the system by its API, it can be executed in any machine; it is not necessary to do it in the same machine where the system is being executed.

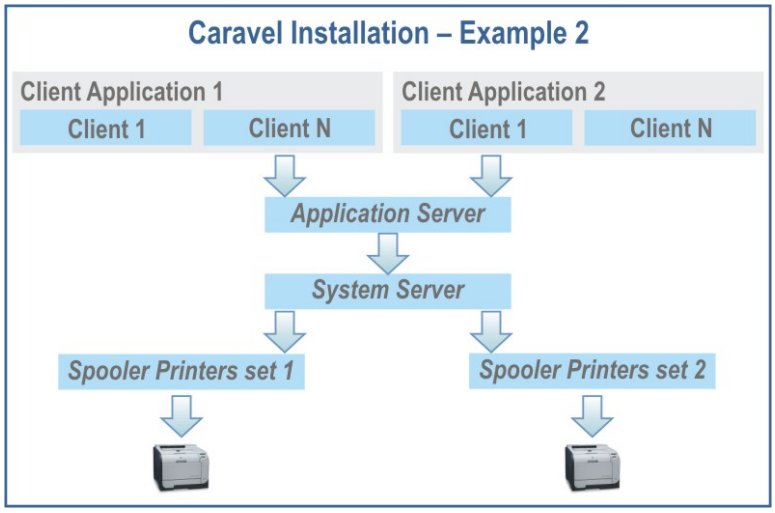
Chapter 10. Installation examples

Combining the client type, the number of system servers required or the set of printers to define, there are different ways of installing an application with access to the *Caravel*.

The following are some examples of possible installations:



In this chart, two different applications are executed in N clients. Each of them has access to a server of a different system. There are also two spooler processes in execution; each of them manages all the printers of one of the systems.



Here we have an application server with two applications executed by think clients. From the application server there is access to the system, shared by the two applications, although it could be different. To manage the printing process two *spooler* processes are activated, one for each set of defined printers; these processes could be executed in one machine, that could be the same of the system server, but they can also be executed in different machines.