Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Skeleton di componenti paralleli riconfigurabili su griglia computazionale Farm e Map

A. Machì, F. Collura

RT-ICAR-PA-03-12 **dicembre 2003**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Skeleton di componenti paralleli riconfigurabili su griglia computazionale

Farm e Map

A. Machì[1], F. Collura[2]

CNR/MIUR Legge 449/97 (5% 1999)
Piattaforma abilitante complessa ad oggetti distribuiti e ad alte prestazioni

Task 1 : Ambiente di programmazione portabile a componenti parallele
Livello L1: Supporti ad Alte prestazioni

Rapporto Tecnico N.: **RT-ICAR-PA-03-12**

Data: **dicembre 2003**

---

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sezione di Palermo
[2] Tesista Università degli Studi di Palermo Dipartimento di Ingegneria Informatica

Indice

1. Introduzione

1.1 Contesto della ricerca

Il progetto CNR-MIUR-***Piattaforma abilitante complessa ad oggetti distribuiti e ad alte prestazioni*** [1].si pone come obbiettivo di "integrare in una stessa piattaforma ITC abilitante complessa le caratteristiche e gli strumenti dell'elaborazione distribuita, del modello a componenti, e del calcolo ad alte prestazioni secondo un approccio unificante. Dal punto di vista dell'architettura fisica, la piattaforma ITC è costituita da un sistema distribuito, realizzato con tecnologia di networking disponibile, i cui sistemi ospiti includono sistemi paralleli ad alte prestazioni, in generale eterogenei l'uno rispetto all'altro". Su tale base il progetto "intende realizzare e sperimentare una piattaforma innovativa a tutti i livelli sovrastanti, che recepisca sia le caratteristiche di modularità, riusabilità e portabilità del modello a componenti che quelle di performance del modello ad alte prestazioni.".

La figura 1 mostra la strutturazione del progetto in tasks esecutivi e la organizzzione del software in 5 livelli di astrazione"

Il Task 2 di tale progetto si propone lo sviluppo di un ambiente di programmazione portabile a componenti parallele **"**per la programmazione di applicazioni HPC in modo indipendente dall'architettura fisica, integrando in un contesto parallelo ambienti e strumenti standard sequenziali e distribuiti. La metodologia è basata sull'integrazione di programmazione parallela strutturata e programmazione a componenti. È nel supporto, compilativo e run-time, che vengono sviluppati componenti per l'ottimizzazione delle prestazioni in modo adattabile alla macchina sottostante.

L'ambiente di programmazione è stratificato in un ambiente di sviluppo vero e proprio ed in un supprto a run time ad alte prestazioni.

Il supporto a run time mette a disposizione tutta una serie di meccanismi che possono essere utilizzati per la realizzazione di un ambiente di programmazione ad alte prestazioni strutturato secondo il paradigma di programmazione ad oggetti. In particolare, il livello L1 mette a disposizione meccanismi per:

- la realizzazione di reti di processi/thread
- lo scheduling di processi/thread sui nodi dell'architettura fisica
- la comunicazione fra processi/thread (punto a punto o collettiva)
- la condivisione di informazione/dati fra processi/thread
- la realizzazione di template (ovvero di forme di cooperazione predefinite, parallele e/o distribuite ed efficienti)

la fault tolerance e la sicurezza, Il tutto con caratteristiche che permettano lo sfruttamento di questi meccanismi per lo sviluppo di applicazioni ad alte prestazioni su architetture che vanno dal cluster di macchine omogenee alla rete di macchine disomogenee distribuite su scala geografica [1].

Il presente rapporto tecnico descrive lo sviluppo (per l'ambiente di supporto a run time) di componenti di libreria ad alte prestazioni scalabili ed adattabili alla configurazione della piattaforma distribuita .

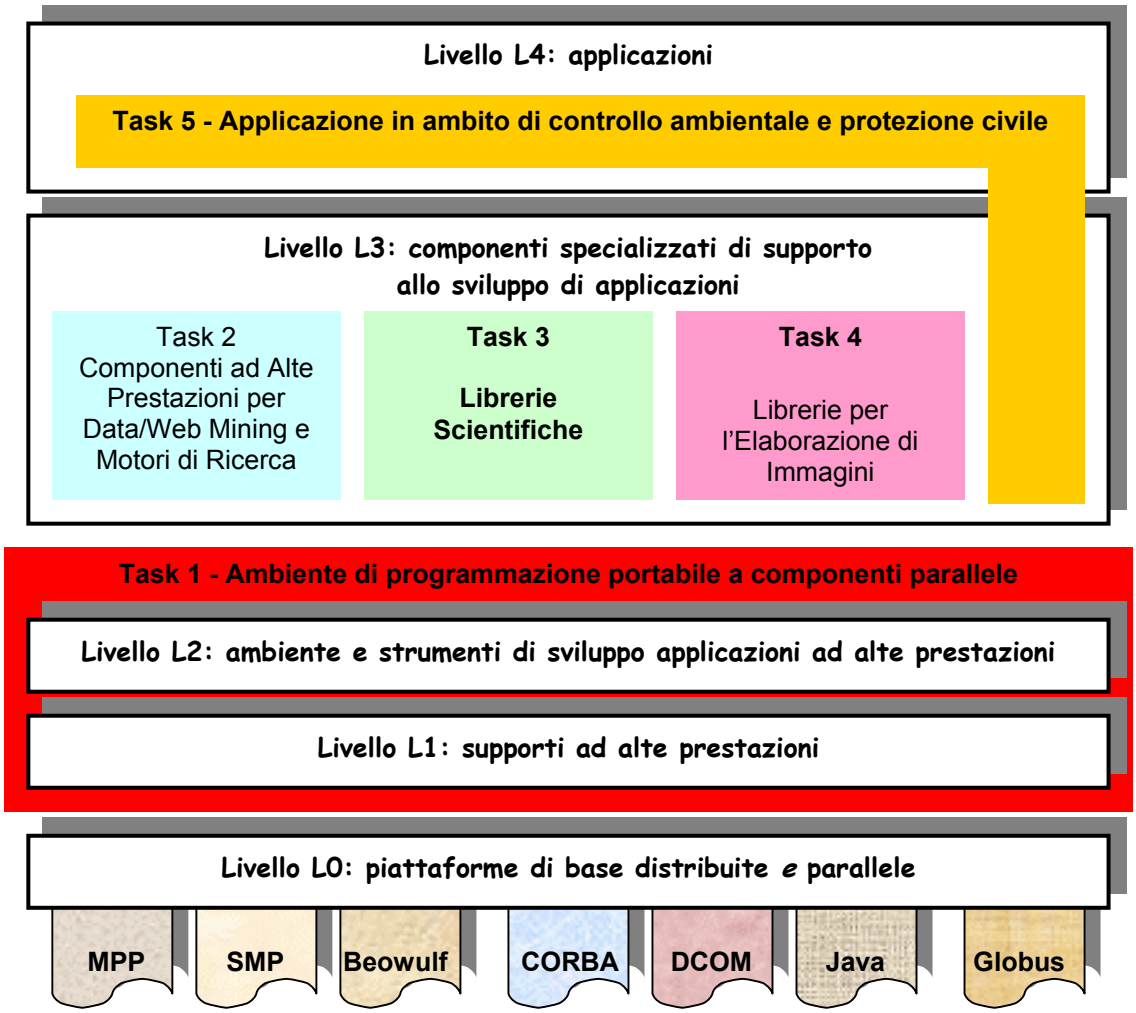


Fig. 1 task del progetto ed organizizzazione gerarchica del software

In particolare l'attività si concentra sullo sviluppo di template implementativi di forme di parallelismo strutturato configurabili dinamicamente a tempo di esecuzione a seconda dell'allocazione dei processi componenti sui nodi ospiti di un sistema distribuito, del grado di eterogeneità delle risorse coinvolte, della loro cardinalità, e della qualità della interconnessione fre i nodi di elaborazione che ospitanoi processori della macchina virtuale di elaborazione distribuita.

1.2 Programmazione parallela strutturata ed ottimizzazione del codice.

Nel corso delle attività di ricerca collegate al progettoPQE2000 [2], sono stati utilizzati i paradigmi della programmazione parallela strutturata, ed in particolare i modelli di skeleton, per predeterminare, a tempo di compilazione (ASSIST-CL)[3], distribuzioni ottimali del carico di lavoro su set di processori virtuali omogenei in base a schemi di cooperazione congeniali alla applicazione.
E' stato finora lasciato al programmatore di effettuare il mapping finale fra set processori virtuali e set di processori fisici, attraverso semplici tools del sistema di controllo della esecuzione (es ASSISTrun) in maniera simile al modello MPI.

Dei tre criteri guida all'ottimizzazione del codice assunti: *Programmabilità, Portabilità, e Performance*, il primo ha portato ad una strutturazione e formalizzazione del modello e degli strumenti di programmazione (uso di skeleton e pattern, definizione di linguaggi di coordinamento), il secondo all'adozione di schemi di strutturazione del software (macchine astratte, processi virtuali, moduli paralleli e componenti), il terzo alla definizione di modelli di componenti generali (parmod) e di schemi di composizione (grafi di processo) basati su modelli di costo conosciuti a tempo di compilazione, validi quindi su architetture distribuite stabili quali cluster di processori o sistemi SMP. In particolare l'attenzione è stata volta, nel PQE2000 a modelli con architetture omogenee o con sottosistemi omogenei accoppiati.[4]

Lo sviluppo delle griglie computazionali e di middleware per la attivazione dinamica di processi e di canali di comunicazione su sistemi remoti con accesso uniforme e sicuro a risorse computazionali e dati (es. GLOBUS Toolkit 2) ha aperto la possibilità di estendere il dominio di applicazione delle metodologia HPC a sistemi diamicamente distribuiti su scala geografica e gestiti da organizzazioni diverse (Organizzazioni Virtuali) [5] .
Il modello OGSA [6] fornisce inoltre un paradigma computazionale per la pubblicazione, scoperta ed attivazione dinamica di servizi computazionali (eventualmente internamente paralleli) su scala mondiale, estendendo ai servizi computazionali su griglia il modello di interazione fra oggetti CORBA [7-8]

Per l'approccio HPC a programmazione parallela strutturata sopra citato, il paradigma di Organizzazione Virtuale, base della fisiologia del modello di Griglia, costituisce un limite alla realizzazione di servizi affidabili, fault- tolerant, con performance prevedibile e con qualità di servizio garantita, in quanto assunzioni base della fisiologia della griglia sono la non stabilità del sistema, e la disponibilità dinamica delle risorse e dei servizi (mantenuti in vita con continui messaggi di keep-alive).
Nel contesto delle griglie computazionali, una piattaforma computazionale è assemblata come Organizzazione Virtuale di risorse su griglia, ed i modelli di costo utilizzati nella programmazione strutturata per selezionare o implementare gli schemi di cooperazione (skeleton) hanno la validità statistica non superiore a quella degli agli indici di qualità delle risorse diponibili.
I meccanismi di preallocazione delle risorse infatti non garantiscono in genere la esclusività del loro uso d'uso (%CPU , banda dei canali di connessione), per il solo per lo staging dei file sono stati sviluppati servizi specifici di griglia (DATAGRID).

E' quindi difficile pensare di richiedere ad una piattaforma di mantenere *QoS* in termini di un tempo limite di esecuzione o di una banda garantita su di un canale per un tempo superiore a quello di stabilità del sistema stesso.

Il *Contratto di Performance* può esseree alaternativamente inteso come una Carta di Servizi offerta dalla piattaforma, legata alle effettive potenzialità assicurabili dalla piattaforma per la esecuzione di un componente software attraverso meccanismi di gestione ottimizzata di un pool di risorse ridondanti allocate semi-dinamicamente sulla griglia. Più che una effettiva misura assoluta di performance, il Contratto garantisce una modalità di scheduling delle richieste di istanze (eventualmente multiple) del componente e la "selezione del grado e della modalità di parallelismo".

Tipici Contratti di Performance possono essere: *servizio in tempo minimo* (es. esecuzione sul massimo di risorse disponibili indipendentemente dall'efficienza del loro uso, al costo conseguente) a supporto di applicazioni interattive o *servizio a costo minimo* (es. esecuzione su un pool ridotto di risorse che ottimiza la efficienza parallela, es al ginocchio della curva di speed-up) per applicazioni off-line. Opzioni di tali contratti possono essere *restart in caso di failure* di un nodo *o di notifica* delle eccezioni.
Il Contratto di performance può essere negoziato con la piattaforma allla installazione del servizio computazionale, utilizzato come criterio per la gestione delle risorse ed eventualmente aggiornato in caso di failure ripetuta delle risorse.

Base della possibilità di rispetto del contratto è la prevedibilità della performance di esecuzione di un task paralello. Tale performance dipende da tre componenti:

- la complessità computazionale del modulo esecutivo che costituisce il nucleo del task, (costo del kernel)
- la efficienza di distribuzione del carico di lavoro sul grafo dei processi cooperanti (costo dello skeleton)
- la potenza esecutiva dei nodi reali su cui è mappato il grafo dei processi virtuali e la banda dei loro canali di interconnessione (costo della infrastruttura di griglia).

Non è in genere possibile costruire un modello analitico del costo globale di un task parallelo per mancanza del modello analitico di uno dei fattori.
Ad esempio la complessità computazionale del kernel può essere irregolare o dinamica. Il comportamento della esecuzione può infatti dipendere dal contenuto dei dati in esame oltre che dal valore numerico dei paramatri di chamata. In altri casi può richiedere risorse variabili dinamicamente nel tempo, o per interazione con l'operatore (parameter steering). In altri casi ancora il costo relativo di alcuni processi del grafo di distribuzione non è conosciuto, o la potenza di esecuzione di un nodo fisico è variabile.

Utilizzando un approccio restricted è possibile limitarsi a un sottoinsieme di casi regolari ed individuare condizioni che permettono la fattorizzazione del modello di costo rendendo trattabile la modellazione della performance.

1.3 Skeleton riconfigurabili su griglia computazionale

La Griglia computazionale definisce un modello flessibile , sicuro e coordinato su larga scala per condividere risorse computazionali. Il modello si focalizza sulla soluzione di problemi su larga scala in organizzazioni virtuali multi-istituzionali. [9].

Il Calcolo ad alte prestazioni è stato, invece, tradizionalmente orientato alla ottimizzazione della performance su risorse di esclusiva proprietà, connesse su network locali o anche geografici. L'ottimizzazione sfrutta pesantemente la conoscenza delle politiche di gestione delle risorse (modelli computazionali, pattern di interconnessione, modelli di costo della interazione fra i processori del grafo rappresentante l'applicazione). Il codice sviluppato con tale approccio per ambienti mappati su network di risorse statiche è inefficiente in ambienti mantenuti come Organizzazioni Virtuali su network geografici, su risorse scopribili dinamicamente ed allocabili solo parzialmente. Il concetto di inaffidabilità, insito nel modello di Organizzazione Virtuale cozza con la prevedibilità dei modelli di costo .

Un approccio al problema consiste nello sviluppare pattern di auto-adattività del codice [10]. In tal caso un nodo del grafo di processo è incaricato di mantenere conoscenza statistica dello stato di performance dei nodi della griglia ed adattare secondo criteri di ottimizzazione il pattern di coordinamento fra i componenti della applicazione.

Un'alternativa [11] consiste nell'adottare un approccio di programmazione gerarchica in cui l'adattività all griglia è distribuita fra vari strati dell'ambiente software che giocano ruoli diversi : l'ambiente di coordinamento delle risorse (active resource/execution manager), un middleware di amministrazione delle risorse (proactive resource administrator), uno strato di coordinamento della qualità di servizio dell'applicazione (reactive quality-service coordinator) ed un middleware di amministrazione effettiva dei nodi esecutivi su griglia (passive platform coordinator).
L'adattività è fattorizzata in: a)scoperta e riserva di nodi di griglia e servizi, e definizione del grafo di processi virtuali che implementano l'applicazione parallela; b) mapping ottimale del grafo di processi sui nodi del pool di risoprse preallocate sulla griglia; c) bilanciamento del carico sui nodi assegnati alla esecuzione del grafo d) monitoraggio del set di processi e (re) configurazione delle porte di interconnessione fra i processi che implementano effettivamente il grafo.
In questo rapporto ci occupiamo della realizzazione dei skeleton paralleli map-data-parallel e farm stream-paralell) [10-15] con capacità di auto-bilanciamento del carico La ad attività è basata su una (ri)configurazione del componente ed è realizzata parametrizzando la politica di ripartizione dei dati (map) o di distribuzione degli elementi (farm) sulla cardinalità dei worker disponibili e sulla loro potenza efficace. In fase di configurazione il nodo producer riceve come parametri, insieme agli argomenti di attivazione del componente, il grafo indicante il mapping effettivo dello skeleton sui nodi della piattaforma distribuita ed indici di peso per nodi ed archi del grafo. Il peso del nodo è proporzionale alla sua potenza efficace (potenza di picco per valore medio di uso) al momento della configurazione ed è supposta stabile per il tempo di esecuzione del modulo.Il processo emitter ricava dal grafo un indice di peso equivalente per ogni nodo worker e lo usa per dirigere la strategia di ripartizione dei dati (map) o degli elementi dello stream (farm).

2. Skeleton riconfigurabili per Farm e Map

Il sistema proposto permette la stesura di algoritmi paralleli secondo due schemi notevoli di programmazione parallela strutturata:

1. *Farm*
2. *Map*

Lo schema *Farm* modella una versione ristretta del paradigma di parallelizzazione master-slave di un flusso di task (stream parallel). Lo schema Map modella una versione ristretta del paradigma di parallelizzazione master-slave di dati compositi (data-parallel)

In entrambi gli schemi si suppone che un *flusso di dati* (stream) debba essere elaborato da un determinato insieme di *unità di esecuzione* (processi, thread, …), alcune di controllo e coordinamento, altre di effettiva capacità elaborativa.
Le unità di controllo sono denominate *Producer*, *Emitter*, *Collector*, *Consumer*.
Le unità di lavoro sono denominate *Worker*.

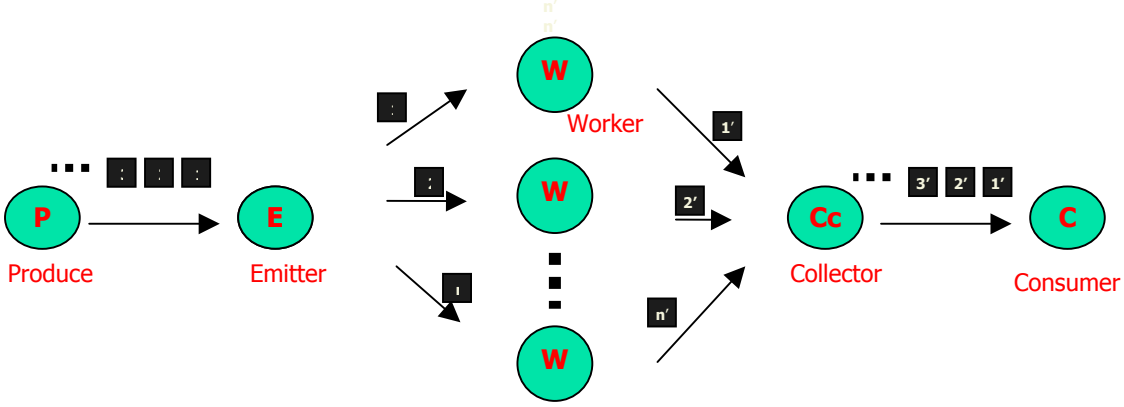
Nel presente sistema, sia le unità di controllo che di lavoro sono dei *processi* in esecuzione su nodi generalmente distinti.

I processi che implementano ogni costrutto sono definiti al di sopra di una macchina virtuale astratta (*implementation template*). la cui struttura (grafo di processi virtuali) è modificabile all'attivazione nella cardinalità dei nodi relativi ai processi di elaborazione
I processi non godono di memoria condivisa. Lo scambio di dati tra processi avviene tramite un opportuno sistema di comunicazione orientato all'inter-scambio di messaggi (quale MPI) con un insieme ridotto di primitive. Il set di primitive permettono la attivazione dei processi, operazioni di trasmissione/ricezione punto punto (send-receive sincrona) e sincronizzazione collettiva (barrier).

Il mapping fra Processori Virtuali e nodi fisici avviene al momemto di attivazione del componente paralelo che implementa il costrutto (configurazione) sulla base di considerazioni di ottimizzazione globale delle risorse computazionali disponibili al momento.
La potenza efficace disponibile su ogni nodo del grafo di mapping e la capacità dei canali di collegamento è conosciuta a tale momento e supposta costante per l'intera durata di esecuzione del task. Tale informazione è fornita al componente parallelo insieme agli altri parametri di attivazione.

3. Skeleton Farm

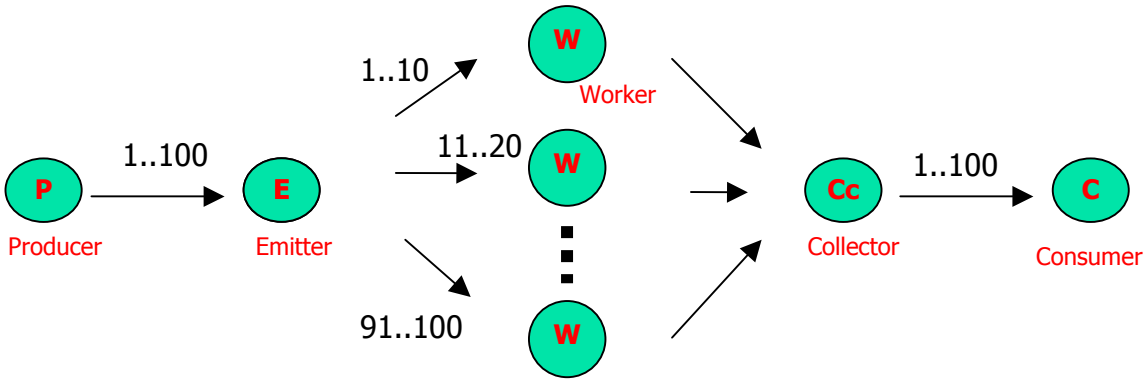
Lo schema Farm effettua una parallelizzazione sullo stream. Ciascun Worker si occuperà dell'elaborazione di una porzione assegnata dello stream di dati.



Il processo ***Producer*** si occupa della generazione dei dati dello stream o in generale di un sequenza ordinata in relazione biunivoca con i reali dati da elaborare.
Il processo ***Emitter*** si occupa di inviare a ciascun Worker un dato da elaborare.
Ciascuno ***Worker*** elabora il dato di pertinenza.
Il processo ***Collector*** si occupa della raccolta dei risultati da parte di ogni Worker, generando quindi uno stream di dati in uscita.
Il processo ***Consumer*** si occupa dello smaltimento dei dati in uscita.

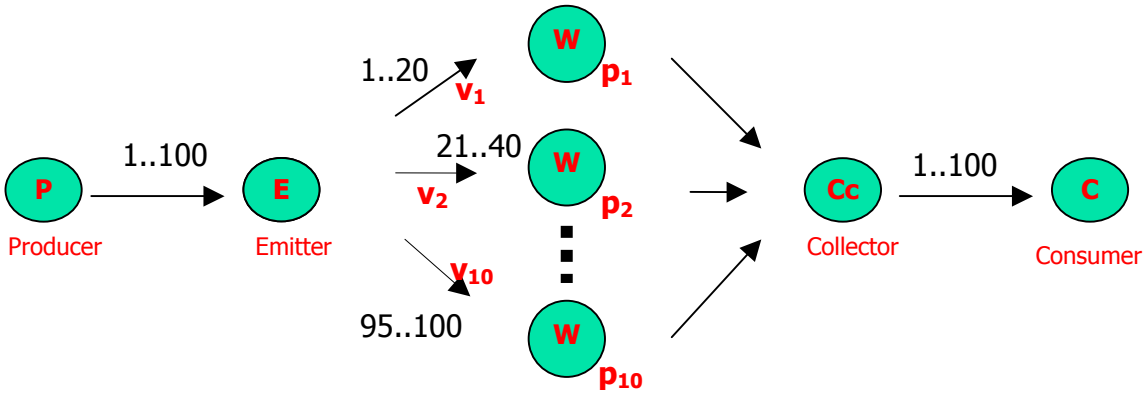
Lo scenario di funzionamento su di una rete **omogenea** nei nodi e nei collegamenti, con un'ipotetica elaborazione di uno stream di *100* dati avendo a disposizione *10* Worker, è il seguente:

1. Lo stream di 100 dati viene suddiviso in 10 parti uguali, ciascuna delle quali è assegnata ad un worker. Ogni parte costituisce un sub-stream a se stante.
2. Il producer provvederà a generare in successione i dati appartenenti a ciascuno dei sub-stream.
3. Ogni dato in arrivo all'emitter viene inviato al worker di pertinenza
4. ogni worker riceverà in sequenza i propri dati da elaborare e provvederà ad inviare i risultati al collector
5. il collector riceverà i risultati in un ordine non predeterminato e provvederà ad inviarli al consumer il quale si occuperà del loro smaltimento



Il sistema proposto prevede, in generale, il funzionamento su di una rete eterogenea in cui ogni nodo è caratterizzato da un indice di performance (pi) ed ogni collegamento è caratterizzato da un indice di velocità (vi). Lo scenario precedente si presenta quando tutti i nodi e tutti i collegamenti sono caratterizzati, rispettivamente, dagli stessi indici di performance e di velocità.

Lo scenario di funzionamento su di una rete **eterogenea** nei nodi e nei collegamenti, con un'ipotetica elaborazione di uno stream di *100* dati avendo a disposizione *10* Worker, è il seguente:



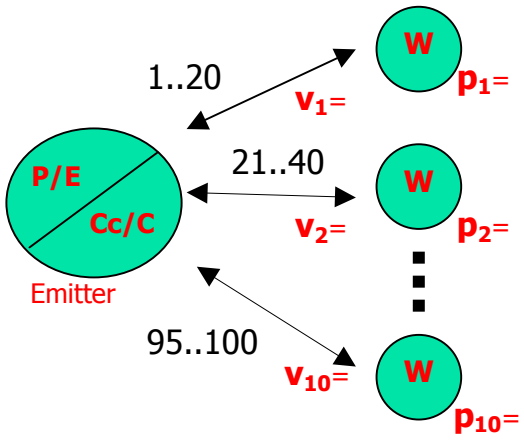
1. A ciascun worker viene attribuito un indice di qualità dipendente linearmente dalla coppia di parametri vi,pi. In particolare, si utilizza la formula Qi=vi+pi
2. Lo stream di 100 dati viene suddiviso in 10 parti, ciascuna delle quali è assegnata ad un worker, in maniera proporzionata al suo indice di qualità Qi. Ogni parte costituisce un sub-stream a se stante
3. Il producer provvederà a generare in successione i dati appartenenti a ciascuno dei sub-stream.
4. Ogni dato in arrivo all'emitter viene inviato al worker di pertinenza
5. ogni worker riceverà in sequenza i propri dati da elaborare e provvederà ad inviare i risultati al collector
6. il collector riceverà i risultati in un ordine non predeterminato e provvederà ad inviarli al consumer il quale si occuperà del loro smaltimento

3.1 Implementazione

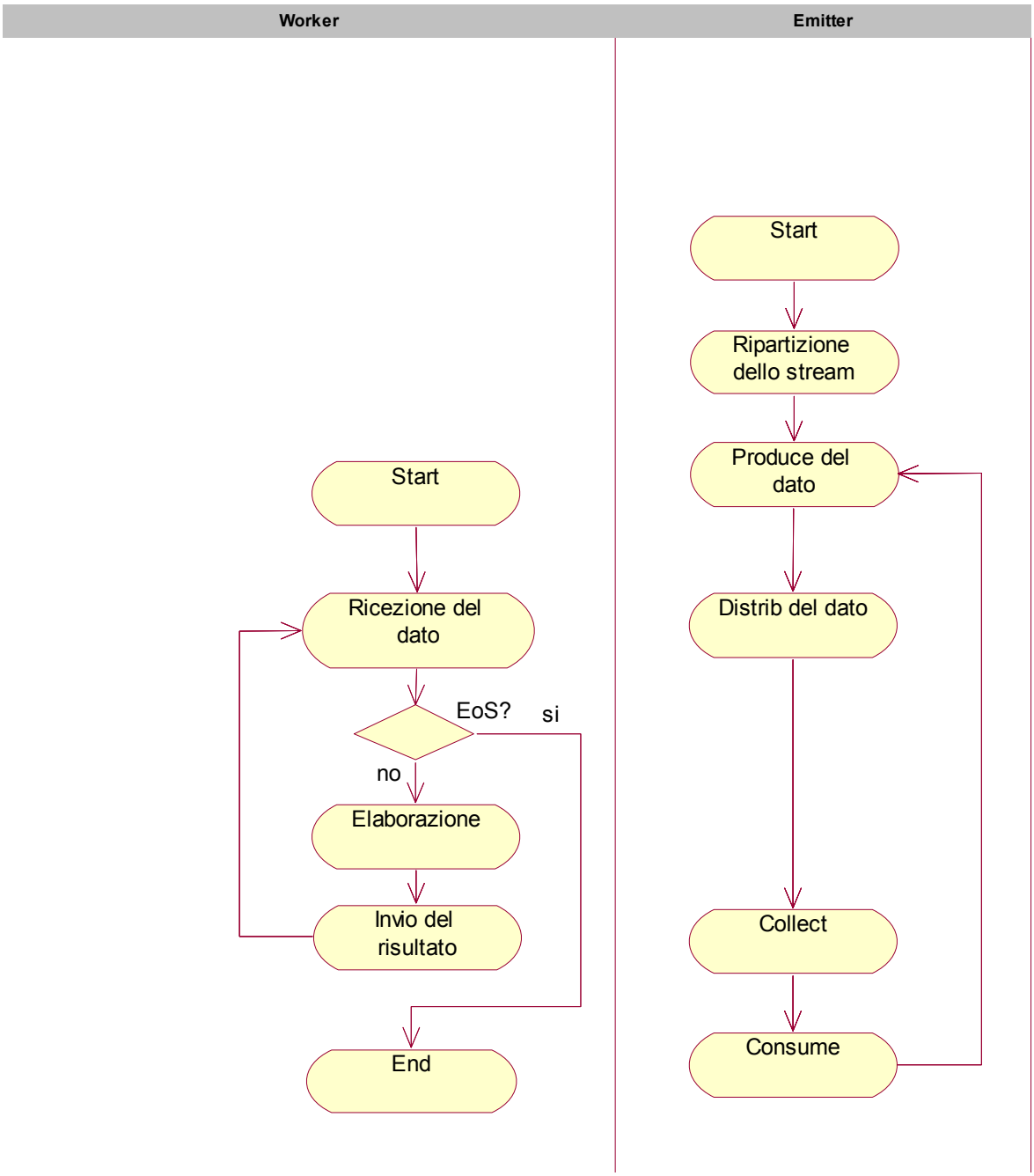
Il sistema proposto implementa lo skeleton Farm in un grafo composto da *n* Worker ed un processo di controllo (Emitter) con mansioni di Producer/Emitter e Collector/Consumer.
Tale processo di controllo effettuerà ripetutamente quattro distinte fasi, corrispondenti alle mansioni dei processi di controllo del grafo logico discusso precedentemente:

- fase di *produce*
- fase di *distrib*
- fase di *collect*
- fase di *consume*

Il funzionamento su di una rete eterogenea nei nodi e nei collegamenti, con un'ipotetica elaborazione di uno stream di *100* dati avendo a disposizione *10* Worker, è il seguente:

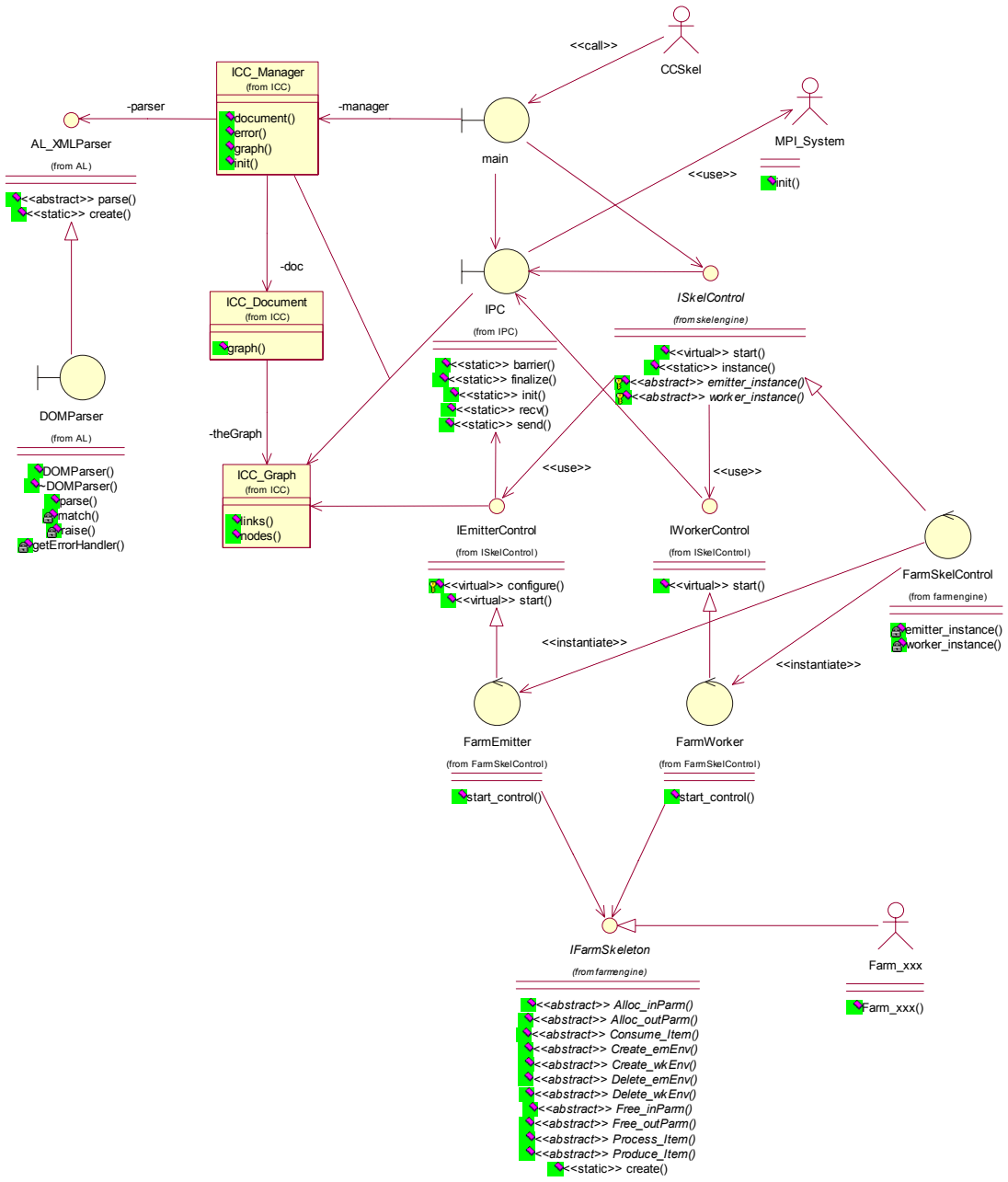


1. A ciascun worker viene attribuito un indice di qualità dipendente linearmente dalla coppia di parametri vi,pi. In particolare, si utilizza la formula Qi=vi+pi
2. Lo stream di 100 dati viene suddiviso in 10 parti, ciascuna delle quali è assegnata ad un worker, in maniera proporzionata al suo indice di qualità Qi. Ogni parte costituisce un sub-stream a se stante
3. L'emitter, nella fase di ***produce***, provvederà a generare in successione i dati di inizio di ciascuno dei sub-stream e provvederà, nella successiva fase di ***distrib***, ad inviarli ai worker pertinenti.
4. ogni worker riceve il proprio dato da elaborare
5. L'emitter entra nella fase di ***collect*** in cui attende il completamento di un'elaborazione da parte di un worker
6. quando un worker termina l'elaborazione invia il risultato nuovamente all'emitter, che nella successiva fase di ***consume*** si appresta a smaltirlo. Successivamente l'emitter genera (fase di produce) il successivo dato appartenente al sub-stream di pertinenza del worker in questione e lo invia (fase di distrib) a quest'ultimo.
7. l'emitter entra nuovamente nella fase di collect (punto 5)
8. quando tutti i dati di un sub-stream vengono correttamente elaborati, l'emitter invia un apposito EoS (end-of-stream) al worker di pertinenza il quale terminerà la sua esecuzione
9. L'intero ciclo termina quando si esauriscono tutti i dati di tutti i sub-stream


Worker
Emitter
Start
Ricezione del dato
EoS?
si
no
Elaborazione
Invio del risultato
End
Start
Ripartizione dello stream
Produce del dato
Distrib del dato
Collect
Consume

3.2 Classi del componente farm

Il componente parallelo che implementa lo skeleton Farm è composto dalle seguenti classi:

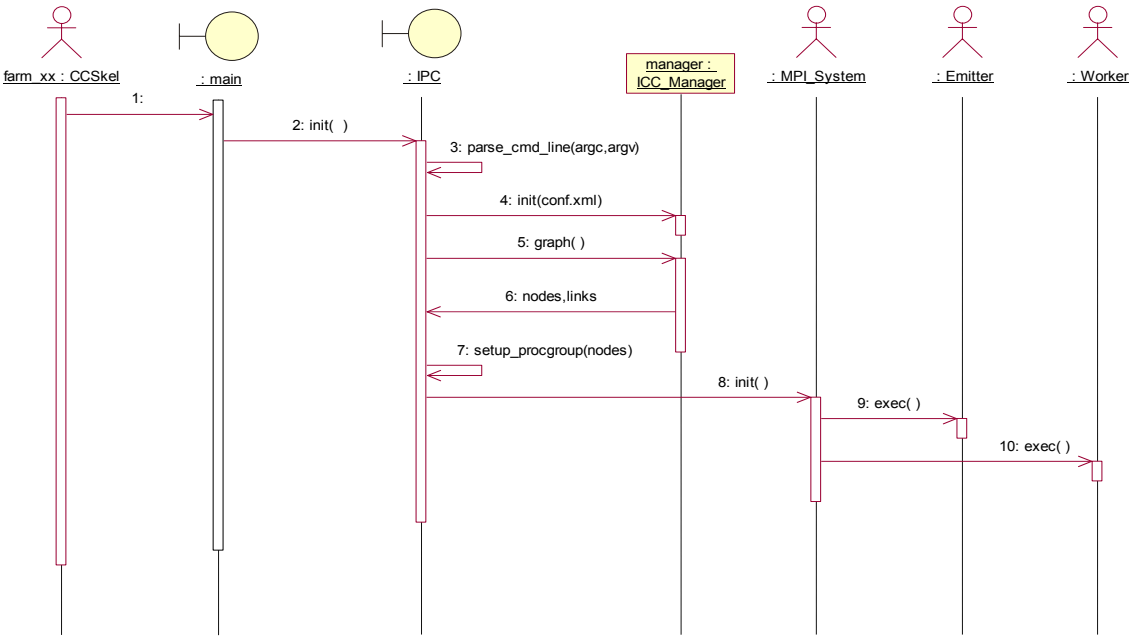


La classe **IPC** fornisce il set RISC di primitive di comunicazione tra processi, implementate tramite l'**MPI_System**.
La classe **ISkeletonControl** definisce l'interfaccia di controllo per un generico skeleton. La classe **FarmSkeletonControl** implementa tale interfaccia per uno skeleton di tipo Farm.
La classe **IFarmSkeleton** definisce l'interfaccia di attivazione di ciascuna delle fasi di un processo di tipo Worker ed Emitter per uno skeleton Farm. Ogni istanza di Farm (**Farm_xxx)** implementa tali fasi secondo l'applicazione specifica.

3.3 Casi d'uso del componente farm

3.3.1 Attivazione – parte 1

Il seguente diagramma di sequenza mostra la prima parte della procedura di attivazione del componente, in cui vengono avviati i processi secondo il grafo di esecuzione specificato.



1-3: Il componente CCSkel inizia la sua esecuzione effettuando il *parsing* della riga di comando. La riga di comando prevista è:

```
farm_xxx -icc <conf.xml> [<arg> …]
```

Il file di configurazione <conf.xml> contiene la composizione del grafo di esecuzione del componente. Tutti i restanti <arg> costituiscono i parametri di attivazione del componente e saranno passati all'Emitter.

4-6: Il componente effettua il parsing del file di configurazione estraendo l'insieme dei nodi e dei link rappresentanti il grafo di esecuzione.

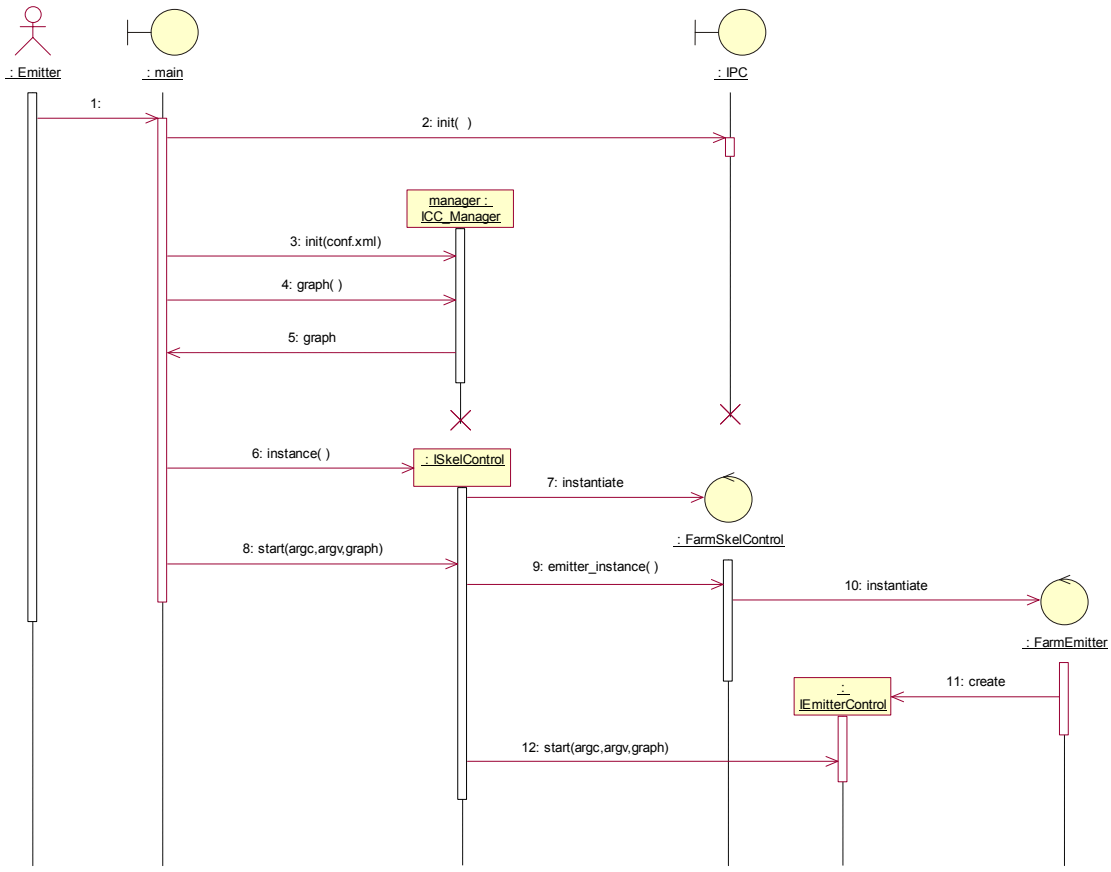
7-8: Sulla base di tali dati, il sistema IPC prepara la specifica del *ProcGroup* di configurazione del sistema MPI e ne invoca l'inizializzazione.

9: Il sistema MPI avvia l'esecuzione dell'Emitter sul nodo pertinente, tramandando i parametri di attivazione <arg> estratti dalla riga di comando.
In realtà, sarà lo stesso processo di partenza a trasformarsi in Emitter. Tale comportamento è determinato dal sistema MPI utilizzato. Come conseguenza diretta si ha che il componente CCSkel dovrà essere attivato su quello che sarà il nodo di esecuzione dell'Emitter.

10: Il sistema MPI avvia l'esecuzione di ogni Worker sui nodi pertinenti.
In questo caso si avrà l'effettiva attivazione di un nuovo processo sul nodo relativo.
Da notare che non necessariamente i processi Worker (o Emitter) dovranno essere mappati su nodi differenti. E' possibile specificare per più Worker ed anche per l'Emitter nodi coincidenti.

3.3.2 Attivazione – parte 2 (Emitter)

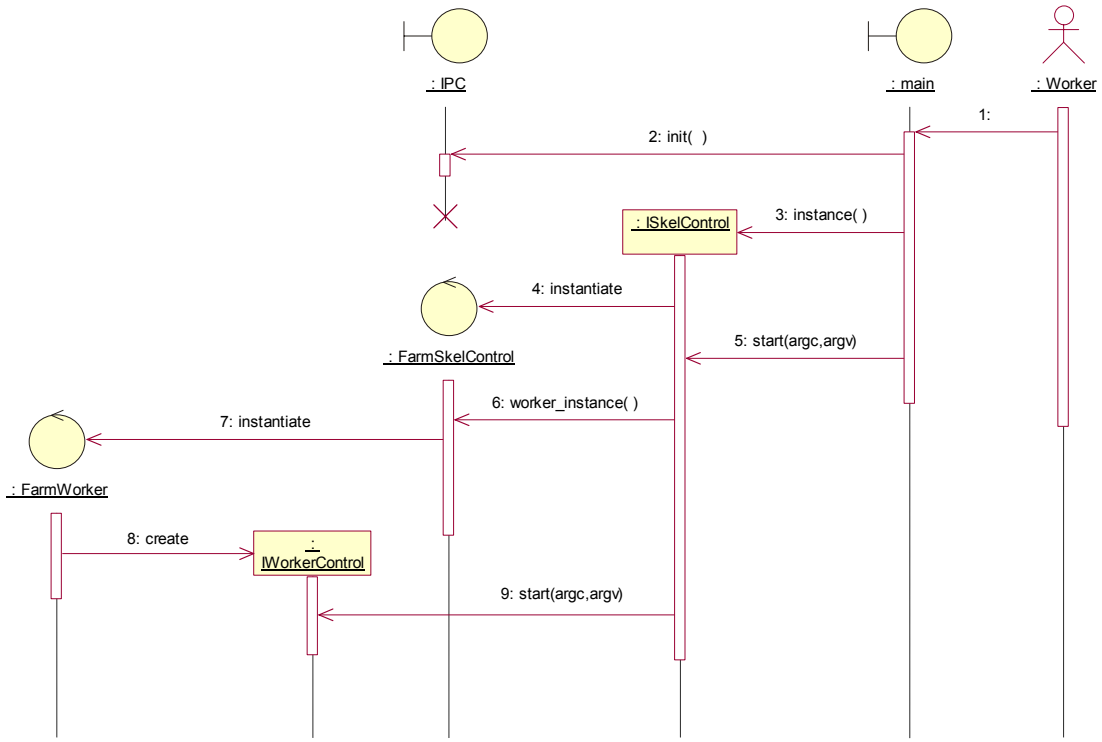
Il seguente diagramma di sequenza mostra la seconda parte della procedura di attivazione del componente, in cui l'Emitter del Farm inizia la sua esecuzione.



1-2: Il processo Emitter inizia l'esecuzione inizializzando il sistema IPC.
3-5: Il processo Emitter estrae dal file di configurazione il grafo di esecuzione.
6: Il processo Emitter richiede un'interfaccia di controllo per lo skeleton.
7: Una classe di controllo di tipo Farm viene istanziata.
8: Il processo Emitter avvia l'esecuzione dello skeleton.
9: L'interfaccia di controllo di tipo Emitter è richiesta.
10-11: Una classe di controllo di tipo Emitter per uno skeleton Farm viene istanziata.
12: L'interfaccia di controllo di tipo Emitter è avviata.

3.3.3 Attivazione – parte 2 (Worker)

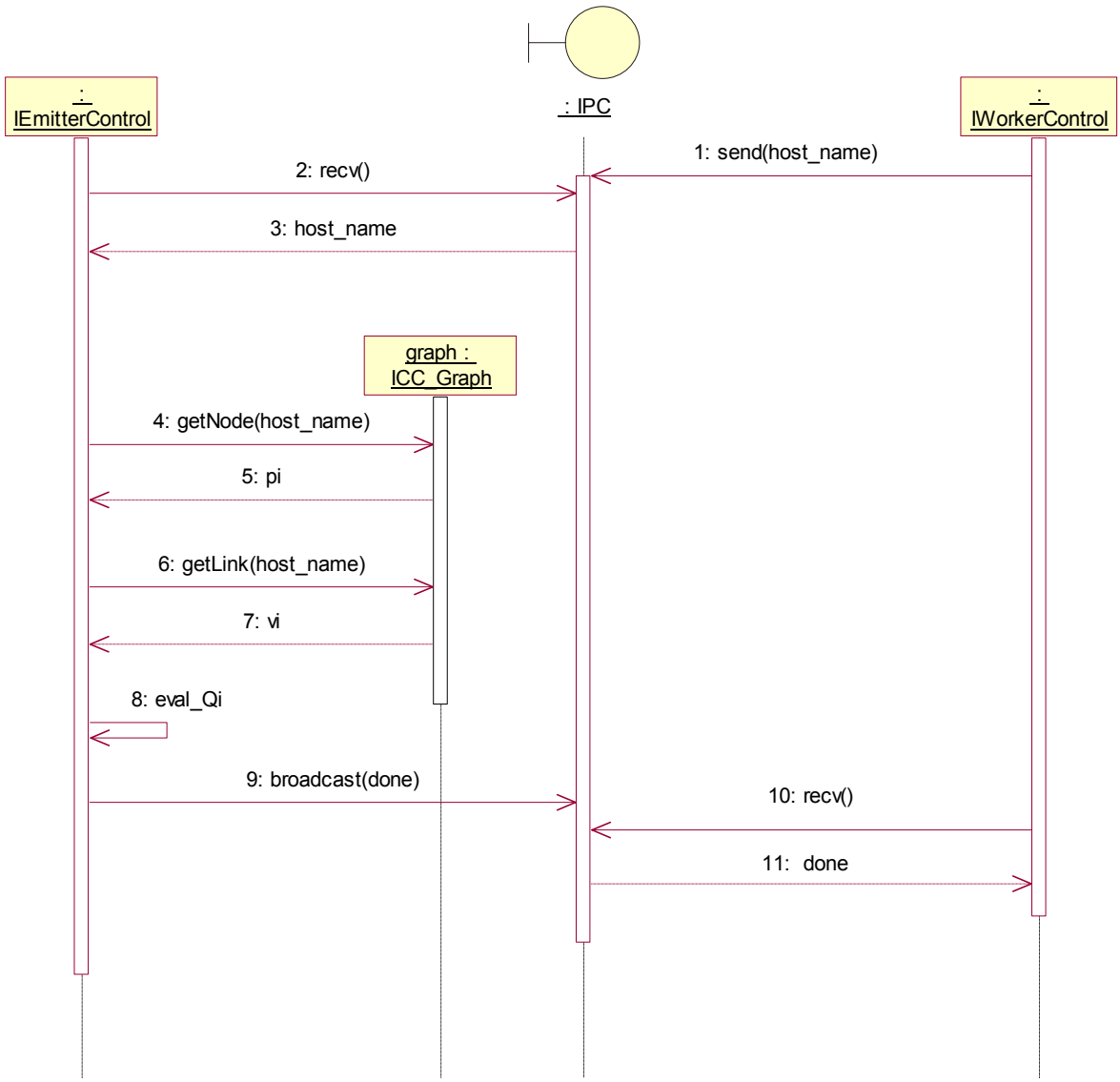
Il seguente diagramma di sequenza mostra la seconda parte della procedura di attivazione del componente, in cui ogni Worker del Farm inizia la sua esecuzione.



1-2: Il processo Worker inizia l'esecuzione inizializzando il sistema IPC.
3: Il processo Worker richiede un'interfaccia di controllo per lo skeleton.
4: Una classe di controllo di tipo Farm viene istanziata.
5: Il processo Worker avvia l'esecuzione dello skeleton.
6: L'interfaccia di controllo di tipo Worker è richiesta.
7-8: Una classe di controllo di tipo Worker per uno skeleton Farm viene istanziata.
9: L'interfaccia di controllo di tipo Worker è avviata.

3.3.4 Configurazione

Il seguente diagramma di sequenza mostra la procedura di configurazione del componente, in cui l'Emitter valuta l'indice di qualità Qi di ogniWorker. Tale procedura è eseguita indifferentemente dal particolare skeleton implementato poichè sfrutta le generiche interfacce di controllo di tipo Emitter e Worker.



1-3: Ogni Worker in esecuzione effettua una registrazione presso l'Emitter, comunicando il proprio nodo di esecuzione.

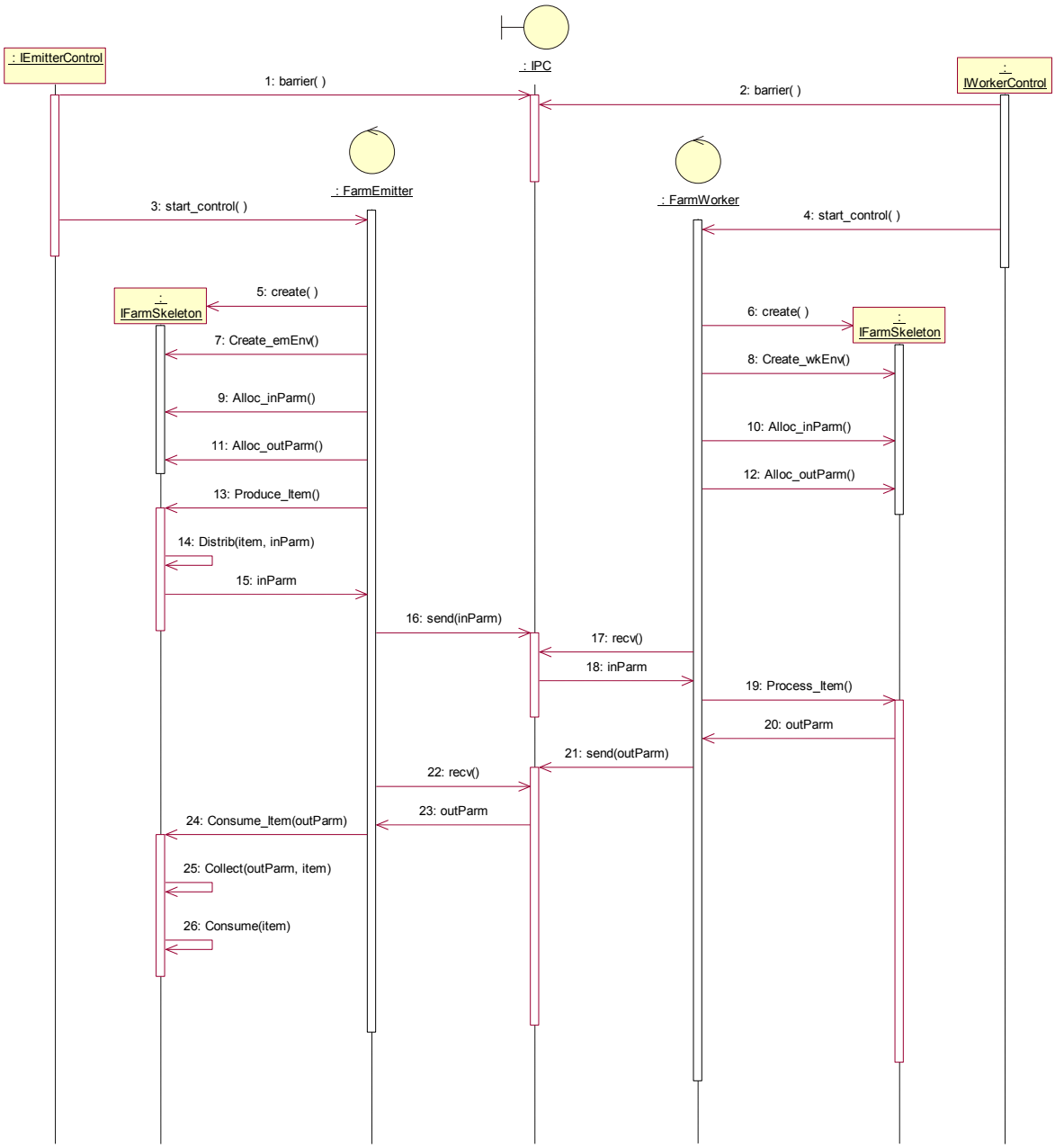
4-7: L'Emitter estrae dal grafo di configurazione gli indici di performance *pi,vi* per ogni nodo registrato.

8: L'Emitter attribuisce gli indici di qualità *Qi* ad ogni Worker.

9-11: La procedura di configurazione viene confermata contemporaneamente a tutti i Worker.

3.3.5 Esecuzione – parte 1

Il seguente diagramma di sequenza mostra la prima parte della procedura di esecuzione del componente, in cui ogni processo si adopera per l'elaborazione parallela effettiva secondo lo skeleton previsto:



1-2: Il generico Worker, terminata la fase di configurazione, attende la sincronizzazione dell'Emitter. L'Emitter provvede a sincronizzarsi con i Worker.

3: L'Emitter inizia l'effettiva esecuzione secondo uno skeleton di tipo Farm.

4: Il Worker inizia l'effettiva esecuzione uno skeleton di tipo Farm.

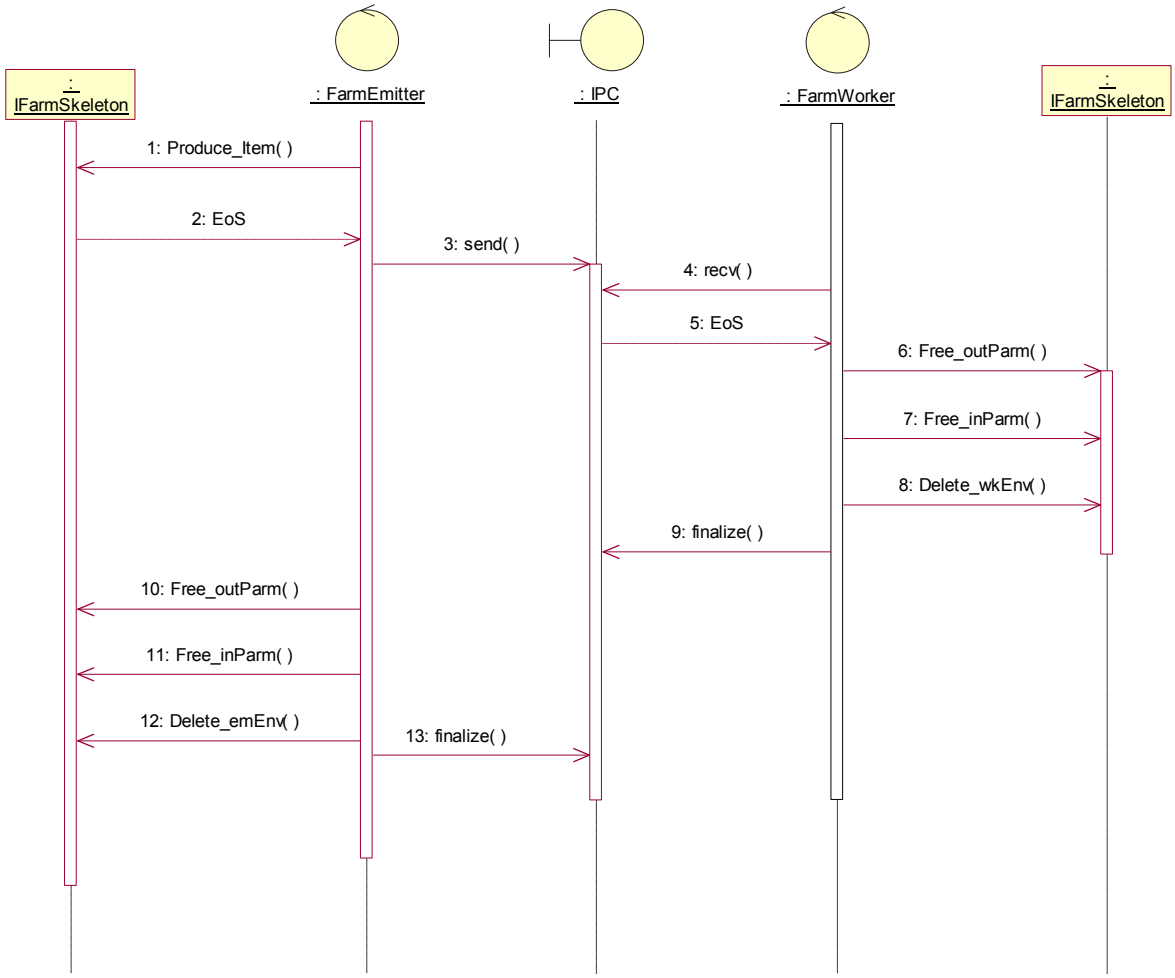
5: L'Emitter crea l'istanza di una classe che implementa le varie fasi di un skeleton di tipo Farm, dipendente dall'applicazione specifica.

6: Ogni Worker crea l'istanza di una classe che implementa le varie fasi di un skeleton di tipo Farm, dipendente dall'applicazione specifica.
7: L'Emitter avvia la fase di creazione dell'ambiente locale.
8: Il Worker avvia la fase di creazione dell'ambiente locale
9: L'Emitter avvia la fase di creazione della struttura dati di Input al Worker, contenente la specifica del generico dato dello stream d'ingresso, dipendente dall'applicazione.
10: Il Worker avvia la fase di creazione della stessa struttura dati di Input dall'Emitter
11: L'Emitter avvia la fase di creazione della struttura dati in Output dal Worker, contenente la specifica del generico dato dello stream di uscita, dipendente dall'applicazione.
12: Il Worker avvia la fase di creazione della stessa struttura dati di OutPut per l'Emitter.
13-15: L'Emitter avvia la fase di ***produce***, generando un dato in Input al Worker.
*La fase di produce termina con la fase di **distrib**, in cui l'Emitter prepara il dato da inviare al Worker secondo modalità dipendenti dall'applicazione specifica.*
16: L'Emitter invia tale dato al Worker.
17-18: Il Worker riceve tale dato dall'Emitter.
19-20: Il Worker avvia la fase di ***process*** per l'elaborazione del dato ottenuto, generando il dato in Output per l'Emitter.
21: Il Worker invia tale dato all'Emitter.
22-23: L'Emitter riceve tale dato dal Worker.
24-26: L'Emitter avvia la fase di *collect & consume*:
*Nella fase di **collect**, l'Emitter prepara il dato ricevuto dal Worker secondo modalità dipendenti dall'applicazione specifica.*
*Nella fasi di **consume**, l'Emitter smaltisce il dato in OutPut dal Worker.*

L'esecuzione prosegue re-iterando dal passo 13, fino a quando nello stream di ingresso sono presenti dei dati da elaborare.

3.3.6 Esecuzione – parte 2

Il seguente diagramma di sequenza mostra la seconda parte della procedura di esecuzione del componente, in cui un Worker termina la propria parte di dati dello stream d'ingresso:



1: L'Emitter avvia la fase di ***produce***, ma nel sub-stream di pertinenza del Worker non esistono più dati.
2: La fase di produce restituisce un indicatore di Eos (end-of-stream).
3: L'Emitter invia l'indicatore di Eos al Worker.
4: Il Worker attende la ricezione del dato da parte dell'Emitter.
5: Il Worker riceve l'indicatore di Eos.
6: Il Worker avvia la fase di distruzione della struttura dati di Output
7: Il Worker avvia la fase di distruzione della struttura dati di Input
8: Il Worker avvia la fase di distruzione dell'ambiente locale
9: Il Worker termina la sua esecuzione finalizzando il sistema IPC.

Quando tutti i Worker hanno ricevuto la segnalazione di EoS ed hanno terminato la loro esecuzione:

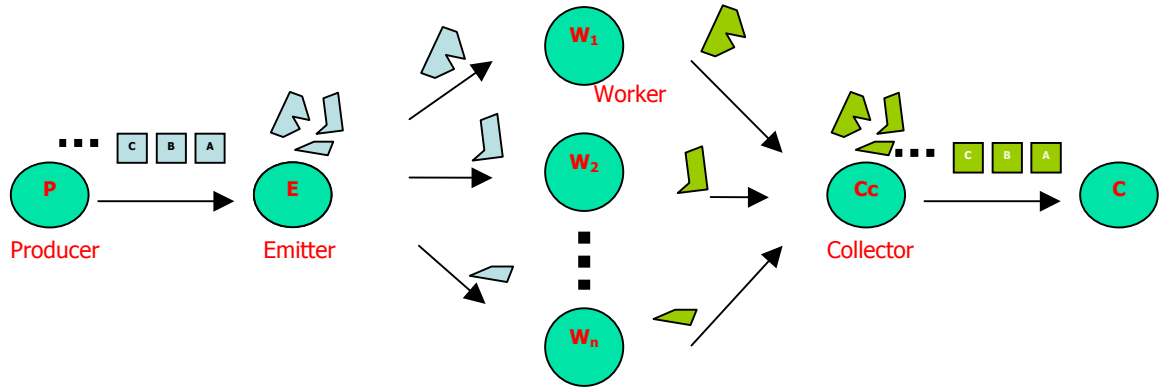
10: L'Emitter avvia la fase di distruzione della struttura dati di Output.

11: L’Emitter avvia la fase di distruzione della struttura dati di Input.
12: L’Emitter avvia la fase di distruzione dell’ambiente locale
13: L’Emitter termina la sua esecuzione finalizzando il sistema IPC.

Il componente termina la sua esecuzione.

4. Skeleton Map

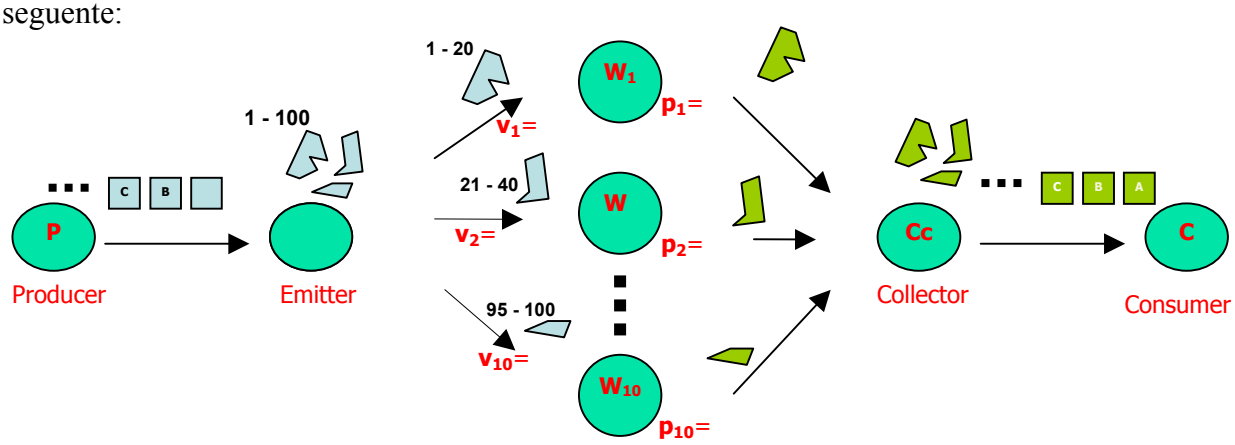
Lo schema Map effettua una parallelizzazione sui dati dello stream. Ciascun Worker si occuperà dell'elaborazione di una porzione assegnata di ogni dato dello stream.



Il processo ***Producer*** si occupa della generazione dei dati dello stream. Il generico dato è supposto divisibile in *n* parti elementari.
Il processo ***Emitter*** si occupa della divisione del dato in più porzioni, ciascuna da inviare ad un Worker per l'elaborazione. Ogni porzione è composta da un certo numero di parti elementari. Le porzioni non sono necessariamente disgiunte ma possono sovrapporsi (overlapping).
Ciascuno ***Worker*** elabora la porzione di pertinenza del dato.
Il processo ***Collector*** si occupa della raccolta dei risultati da parte di ogni Worker e del loro riassemblamento per la costituzione del dato finale, generando quindi uno stream di dati in uscita.
Il processo ***Consumer*** si occupa dello smaltimento dei dati in uscita.

Il sistema proposto prevede, in generale, il funzionamento su di una rete eterogenea in cui ogni nodo è caratterizzato da un indice di performance (pi) ed ogni collegamento è caratterizzato da un indice di velocità (vi).

Lo scenario di funzionamento con un'ipotetica elaborazione di uno stream di dati, ciascuno divisibile in *100* parti elementari, avendo a disposizione *10* Worker, è il seguente:



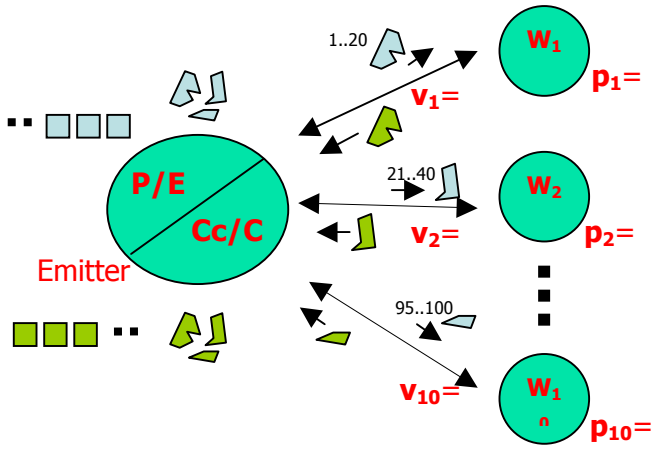
1. A ciascun worker viene attribuito un indice di qualità dipendente linearmente dalla coppia di parametri vi,pi. In particolare, si utilizza la formula Qi=vi+pi.
2. Il producer provvederà a generare in successione i dati dello stream.
3. Ogni dato in arrivo all'emitter è suddiviso in *10* porzioni, ciascuna delle quali è inviata ad un worker e contiene un numero di parti elementari proporzionato all'indice di qualità Qi del worker di pertinenza.
4. Ogni worker riceverà in sequenza le porzioni di dato da elaborare e provvederà ad inviare i risultati al collector
5. Il collector riceverà i risultati dai worker, provvederà al riassemblamento delle porzioni elaborate ottenendo i dati finali da inviare al consumer il quale si occuperà del loro smaltimento

4.2.1 Implementazione

Il sistema proposto implementa lo skeleton Map in un grafo composto da *n* Worker ed un processo di controllo (Emitter) con mansioni di Producer/Emitter e Collector/Consumer.
Tale processo di controllo effettuerà ripetutamente quattro distinte fasi, corrispondenti alle mansioni dei processi di controllo del grafo logico discusso precedentemente:

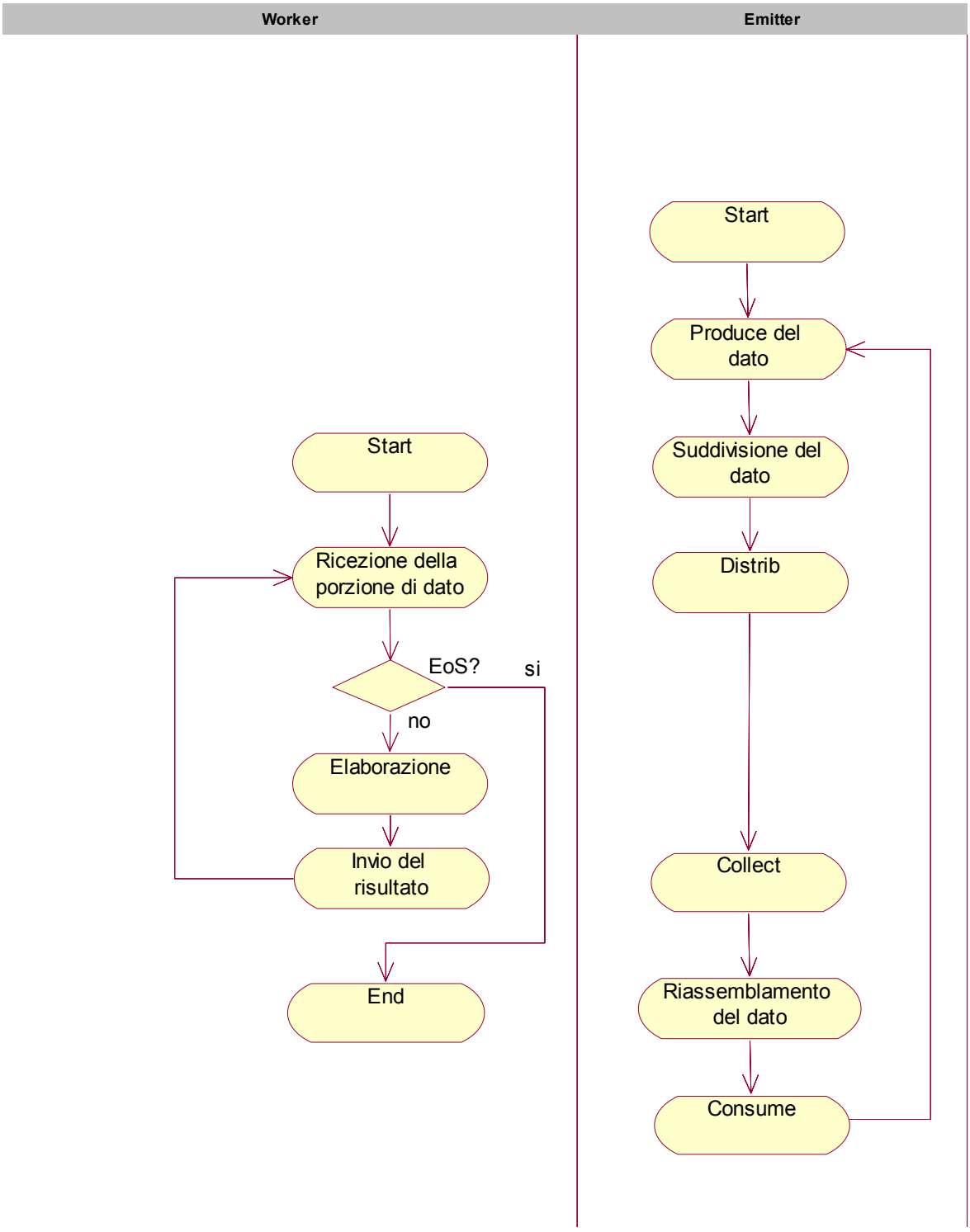
- fase di *produce*
- fase di *distrib*
- fase di *collect*
- fase di *consume*

Il funzionamento con un'ipotetica elaborazione di uno stream di dati, ciascuno divisibile in *100* parti elementari, avendo a disposizione *10* Worker, è il seguente:



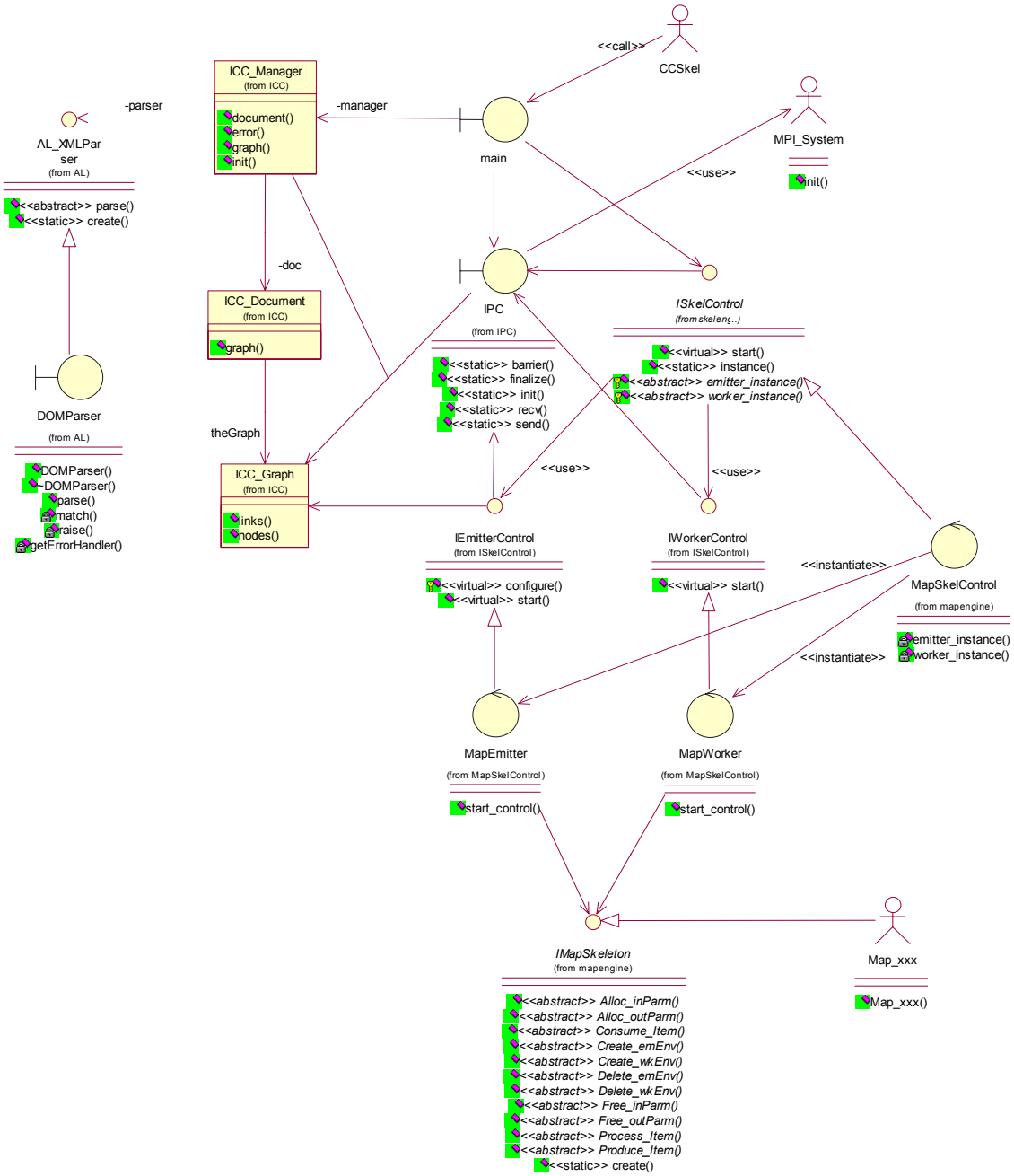
1. A ciascun worker viene attribuito un indice di qualità dipendente linearmente dalla coppia di parametri vi,pi. In particolare, si utilizza la formula Qi=vi+pi
2. L'emitter, nella fase di ***produce***, provvederà a generare un dato dello stream e procederà, nella successiva fase di ***distrib***, alla sua suddivisione in *10* porzioni, ciascuna delle quali sarà inviata ad un worker e conterrà un numero di parti elementari proporzionato all'indice di qualità Qi del worker di pertinenza.
3. Ogni worker riceve la propria porzione di dato da elaborare.

4. L’emitter entra nella fase di ***collect*** in cui attende il completamento dell’elaborazione da parte di ogni worker.
5. Ogni worker che termina l’elaborazione invia la propria parte di risultato nuovamente all’emitter
6. Quando tutti i worker terminano l’elaborazione, l’emitter provvede al riassemblamento del dato finale e procede, nella successiva fase di ***consume*** , al suo smaltimento. Successivamente l’emitter genera (fase di produce) il successivo dato dello stream, procede nella nuova fase di distrib ed entra nuovamente nella fase di collect (punto 4)
7. Quando tutti i dati dello stream vengono correttamente elaborati, l’emitter invia un apposito EoS (end-of-stream) ad ogni worker il quale terminerà la sua esecuzione.


Worker
Emitter
Start
Ricezione della porzione di dato
EoS?
si
no
Elaborazione
Invio del risultato
End
Start
Produce del dato
Suddivisione del dato
Distrib
Collect
Riassemblamento del dato
Consume

4.2 Classi del componente map

Il componente parallelo che implementa lo skeleton Map è composto dalle seguenti classi:



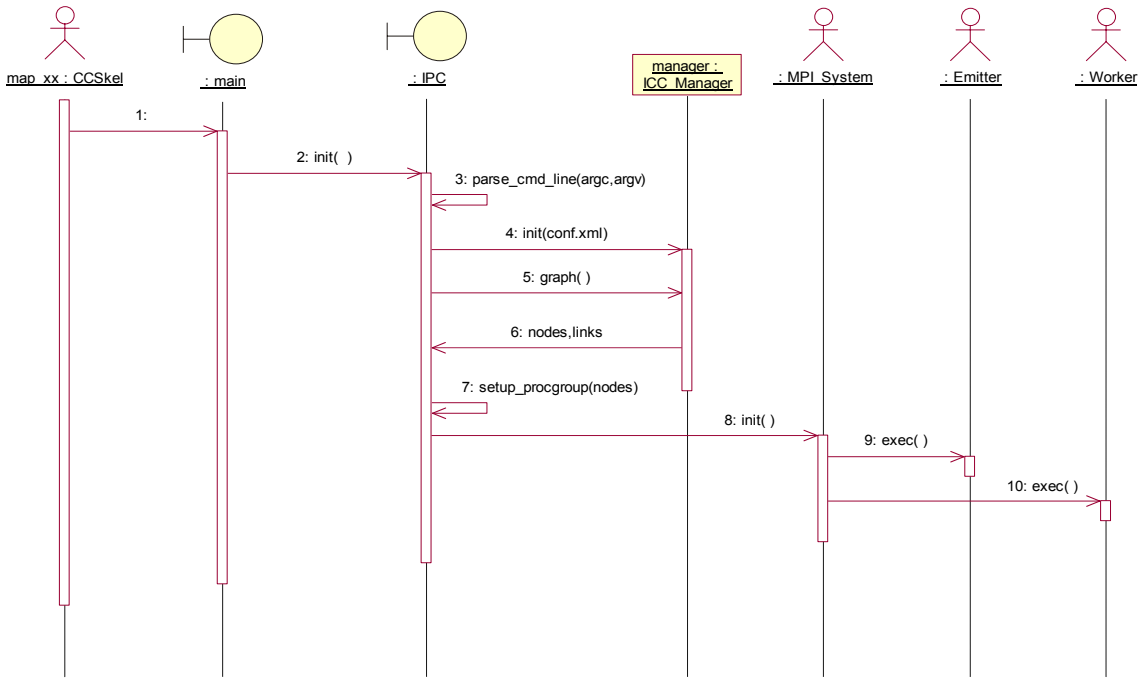
La classe **IPC** fornisce il set RISC di primitive di comunicazione tra processi, implementate tramite l'**MPI_System**.
La classe **ISkeletonControl** definisce l'interfaccia di controllo per un generico skeleton. La classe **MapSkeletonControl** implementa tale interfaccia per uno skeleton di tipo Map.

La classe **IMapSkeleton** definisce l'interfaccia di attivazione di ciascuna delle fasi di un processo di tipo Worker ed Emitter per uno skeleton Map. Ogni istanza di Map (**Map_xxx)** implementa tali fasi secondo l'applicazione specifica.

4.3 Casi d'uso del componente map

4.3.1 Attivazione – parte 1

Il seguente diagramma di sequenza mostra la prima parte della procedura di attivazione del componente, in cui vengono avviati i processi secondo il grafo di esecuzione specificato.



1-3: Il componente CCSkel inizia la sua esecuzione effettuando il *parsing* della riga di comando. La riga di comando prevista è:

```
map_xxx -icc <conf.xml> [<arg> …]
```

Il file di configurazione <conf.xml> contiene la composizione del grafo di esecuzione del componente. Tutti i restanti <arg> costituiscono i parametri di attivazione del componente e saranno passati all'Emitter.

4-6: Il componente effettua il parsing del file di configurazione estraendo l'insieme dei nodi e dei link rappresentanti il grafo di esecuzione.

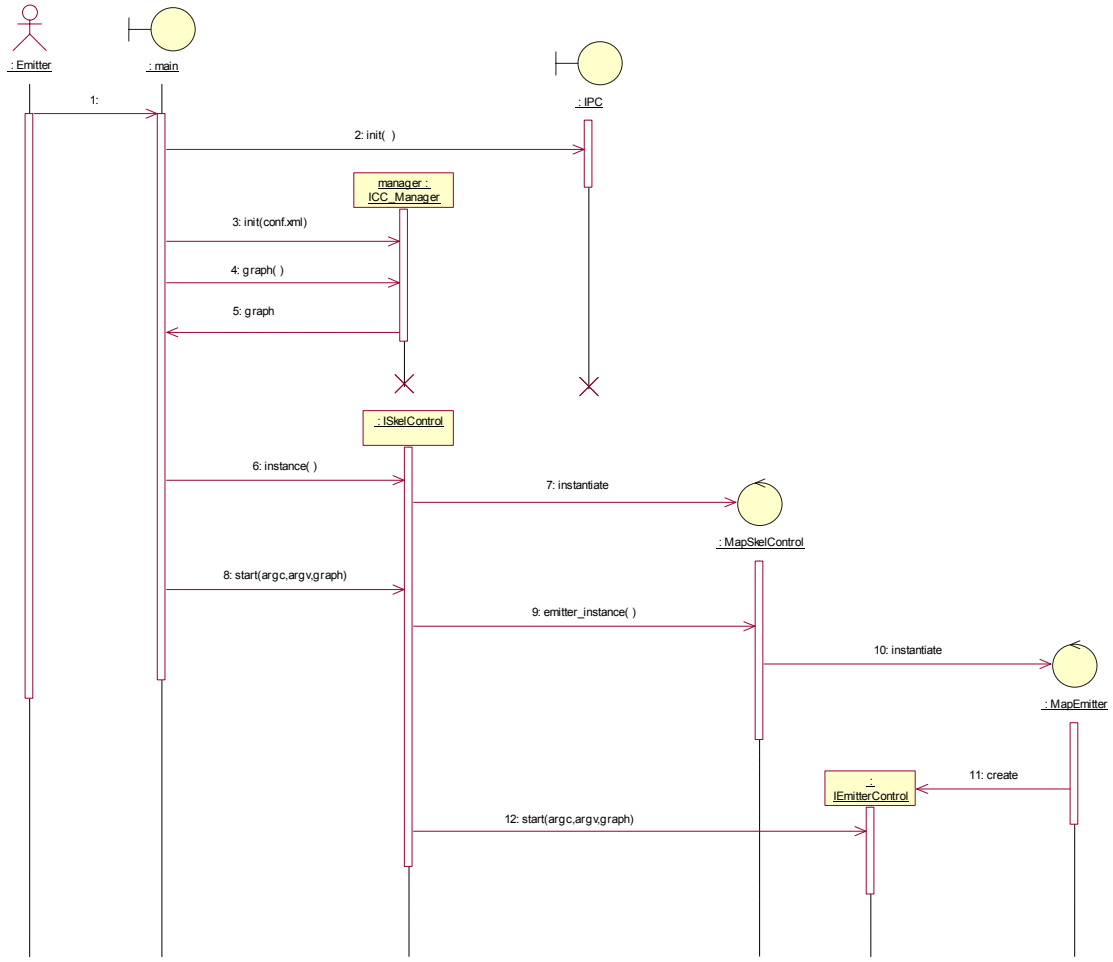
7-8: Sulla base di tali dati, il sistema IPC prepara la specifica del *ProcGroup* di configurazione del sistema MPI e ne invoca l'inizializzazione.

9: Il sistema MPI avvia l'esecuzione dell'Emitter sul nodo pertinente, tramandando i parametri di attivazione <arg> estratti dalla riga di comando.
In realtà, sarà lo stesso processo di partenza a trasformarsi in Emitter. Tale comportamento è determinato dal sistema MPI utilizzato. Come conseguenza diretta si ha che il componente CCSkel dovrà essere attivato su quello che sarà il nodo di esecuzione dell'Emitter.

10: Il sistema MPI avvia l'esecuzione di ogni Worker sui nodi pertinenti.
In questo caso si avrà l'effettiva attivazione di un nuovo processo sul nodo relativo.
Da notare che non necessariamente i processi Worker (o Emitter) dovranno essere mappati su nodi differenti. E' possibile specificare per più Worker ed anche per l'Emitter nodi coincidenti.

4.3.2 Attivazione – parte 2 (Emitter)

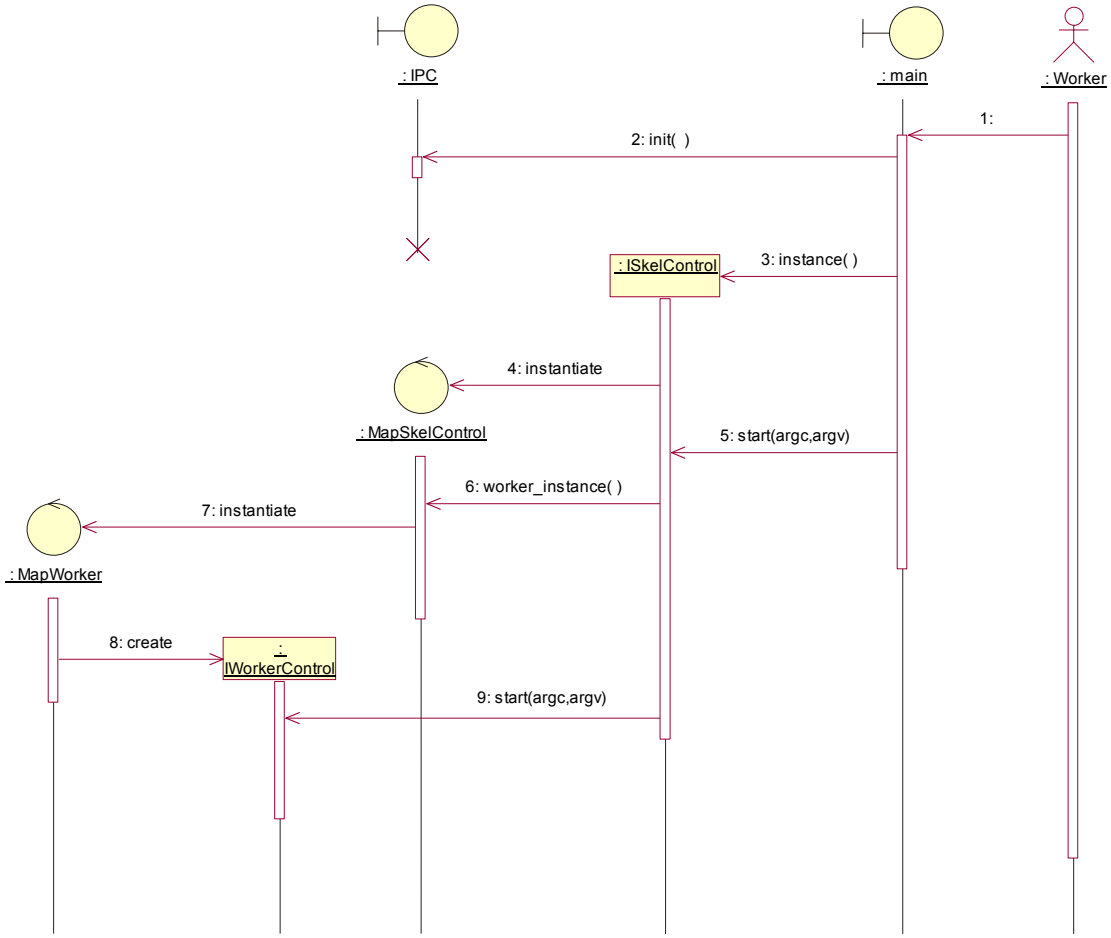
Il seguente diagramma di sequenza mostra la seconda parte della procedura di attivazione del componente, in cui l'Emitter del Map inizia la sua esecuzione.



1-2: Il processo Emitter inizia l'esecuzione inizializzando il sistema IPC.
3-5: Il processo Emitter estrae dal file di configurazione il grafo di esecuzione.
6: Il processo Emitter richiede un'interfaccia di controllo per lo skeleton.
7: Una classe di controllo di tipo Map viene istanziata.
8: Il processo Emitter avvia l'esecuzione dello skeleton.
9: L'interfaccia di controllo di tipo Emitter è richiesta.
10-11: Una classe di controllo di tipo Emitter per uno skeleton Map viene istanziata.
12: L'interfaccia di controllo di tipo Emitter è avviata.

4.3.3 Attivazione – parte 2 (Worker)

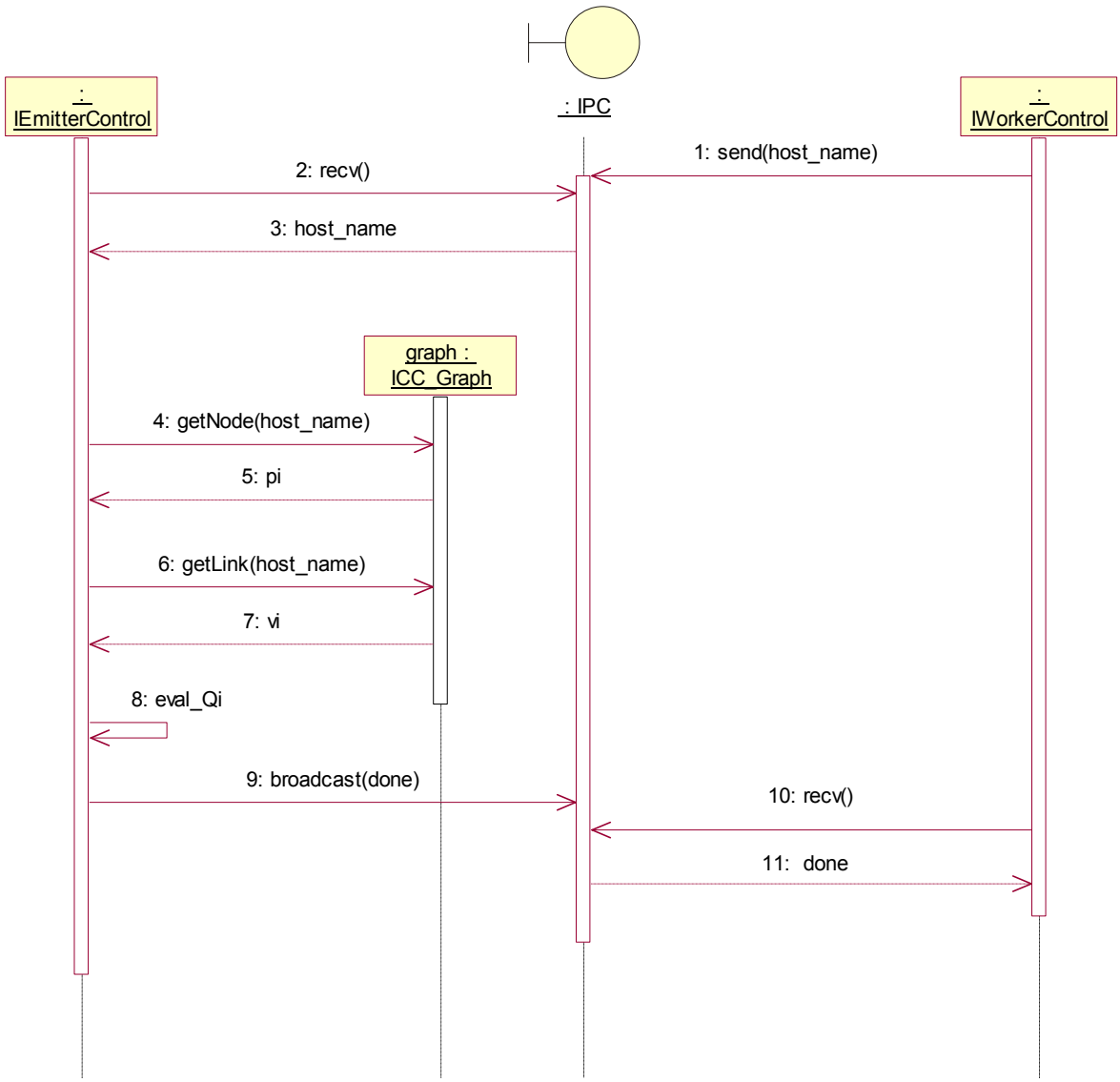
Il seguente diagramma di sequenza mostra la seconda parte della procedura di attivazione del componente, in cui ogni Worker del Map inizia la sua esecuzione.



1-2: Il processo Worker inizia l'esecuzione inizializzando il sistema IPC.
3: Il processo Worker richiede un'interfaccia di controllo per lo skeleton.
4: Una classe di controllo di tipo Map viene istanziata.
5: Il processo Worker avvia l'esecuzione dello skeleton.
6: L'interfaccia di controllo di tipo Worker è richiesta.
7-8: Una classe di controllo di tipo Worker per uno skeleton Map viene istanziata.
9: L'interfaccia di controllo di tipo Worker è avviata.

4.3.4 Configurazione

Il seguente diagramma di sequenza mostra la procedura di configurazione del componente, in cui l'Emitter valuta l'indice di qualità Qi di ogniWorker. Tale procedura è eseguita indifferentemente dal particolare skeleton implementato poichè sfrutta le generiche interfacce di controllo di tipo Emitter e Worker.



1-3: Ogni Worker in esecuzione effettua una registrazione presso l'Emitter, comunicando il proprio nodo di esecuzione.

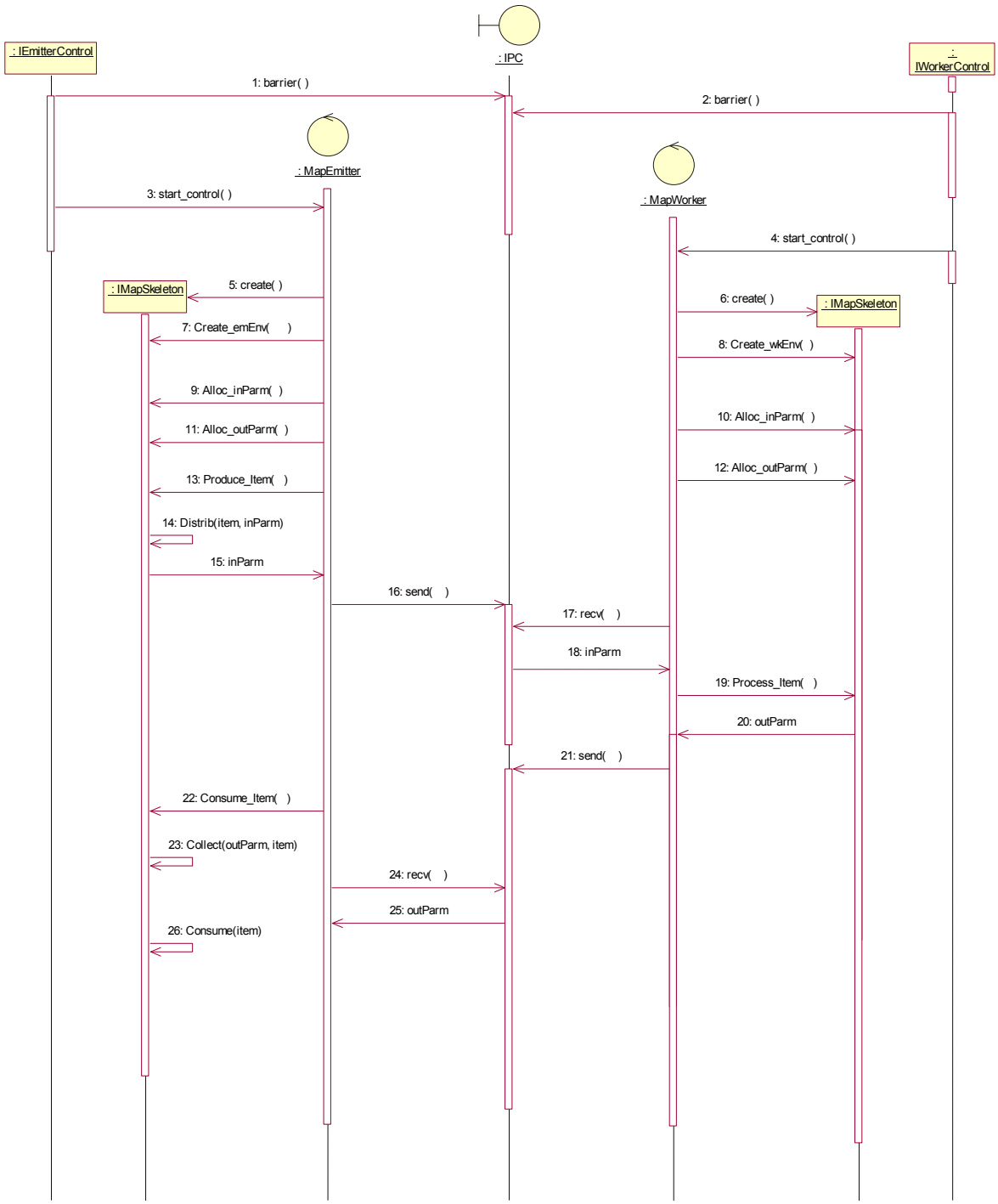
4-7: L'Emitter estrae dal grafo di configurazione gli indici di performance *pi,vi* per ogni nodo registrato.

8: L'Emitter attribuisce gli indici di qualità *Qi* ad ogni Worker.

9-11: La procedura di configurazione viene confermata contemporaneamente a tutti i Worker.

4.3.5 Esecuzione – parte 1

Il seguente diagramma di sequenza mostra la prima parte della procedura di esecuzione del componente, in cui ogni processo si adopera per l'elaborazione parallela effettiva secondo lo skeleton previsto:

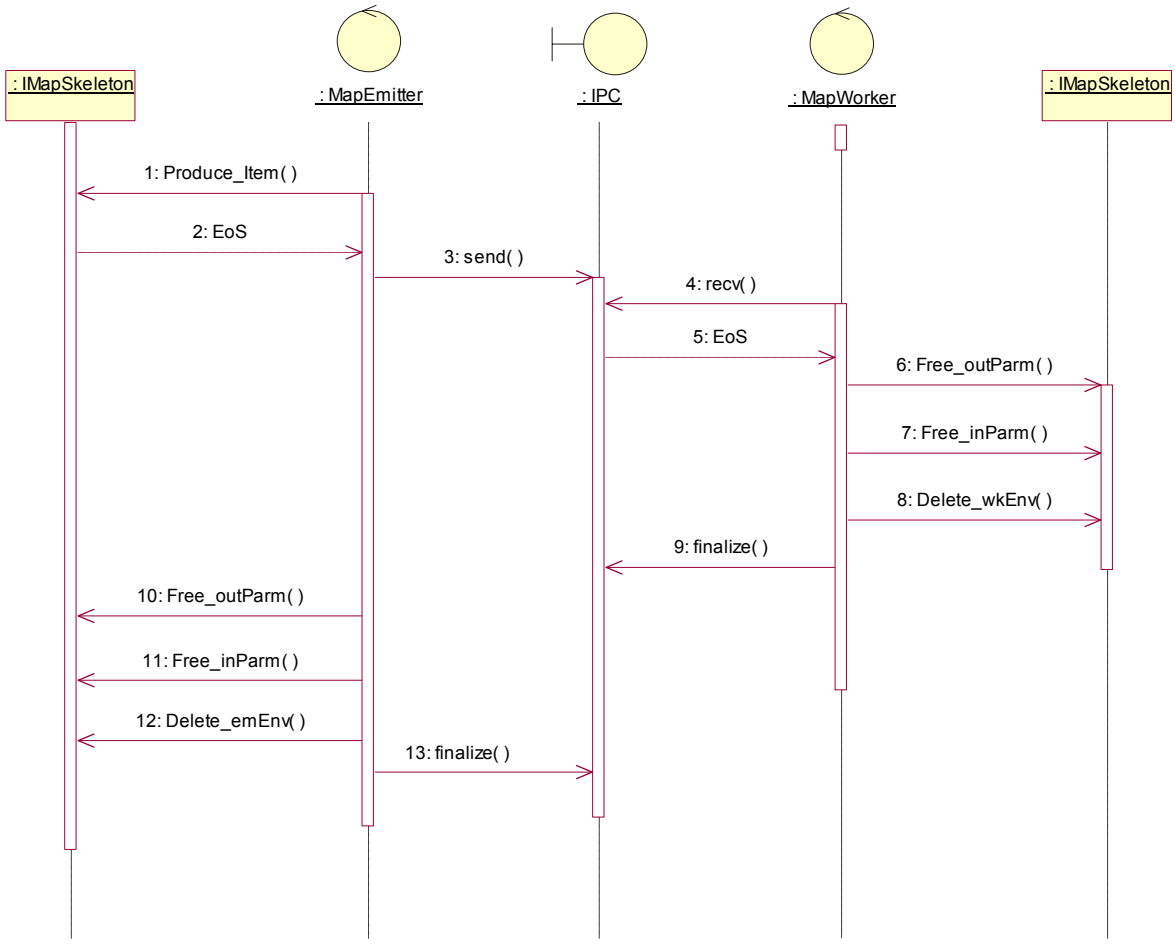


1-2: Il generico Worker, terminata la fase di configurazione, attende la sincronizzazione dell'Emitter. L'Emitter provvede a sincronizzarsi con i Worker.
3: L'Emitter inizia l'effettiva esecuzione secondo uno skeleton di tipo Map.
4: Il Worker inizia l'effettiva esecuzione uno skeleton di tipo Map.
5: L'Emitter crea l'istanza di una classe che implementa le varie fasi di un skeleton di tipo Map, dipendente dall'applicazione specifica.
6: Ogni Worker crea l'istanza di una classe che implementa le varie fasi di un skeleton di tipo Map, dipendente dall'applicazione specifica.
7: L'Emitter avvia la fase di creazione dell'ambiente locale.
8: Il Worker avvia la fase di creazione dell'ambiente locale.
9: L'Emitter avvia la fase di creazione della struttura dati di Input al Worker, contenente la specifica del generico dato dello stream d'ingresso, dipendente dall'applicazione.
10: Il Worker avvia la fase di creazione della stessa struttura dati di Input dall'Emitter
11: L'Emitter avvia la fase di creazione della struttura dati in Output dal Worker, contenente la specifica del generico dato dello stream di uscita, dipendente dall'applicazione.
12: Il Worker avvia la fase di creazione della stessa struttura dati di OutPut per l'Emitter.
13-15: L'Emitter avvia la fase di ***produce***, generando un dato dello stream.
*La fase di produce termina con la fase di **distrib**, in cui l'Emitter prepara la porzione di dato da inviare al Worker secondo modalità dipendenti dall'applicazione specifica.*
16: L'Emitter invia tale porzione del dato al Worker.
17-18: Il Worker riceve tale porzione del dato dall'Emitter.
19-20: Il Worker avvia la fase di ***process*** per l'elaborazione della porzione del dato ottenuta, generando la porzione di dato in Output per l'Emitter.
21: Il Worker invia tale porzione del dato all'Emitter.
22-26: L'Emitter avvia la fase di *collect & consume*:
*Nella fase di **collect**, l'Emitter riceve una porzione di dato da ogni worker. Quando tutti i worker hanno inviato i loro risultati il dato finale viene riassemblato secondo modalità dipendenti dall'applicazione specifica.*
*Nella fasi di **consume**, l'Emitter smaltisce il dato.*

L'esecuzione prosegue re-iterando dal passo 13, fino a quando nello stream di ingresso sono presenti dei dati da elaborare.

4.3.6 Esecuzione – parte 2

Il seguente diagramma di sequenza mostra la seconda parte della procedura di esecuzione del componente, in cui un Worker termina la propria parte di dati dello stream d'ingresso:



1: L'Emitter avvia la fase di ***produce***, ma nel sub-stream di pertinenza del Worker non esistono più dati.
2: La fase di produce restituisce un indicatore di Eos (end-of-stream).
3: L'Emitter invia l'indicatore di Eos al Worker.
4: Il Worker attende la ricezione del dato da parte dell'Emitter.
5: Il Worker riceve l'indicatore di Eos.
6: Il Worker avvia la fase di distruzione della struttura dati di Output
7: Il Worker avvia la fase di distruzione della struttura dati di Input
8: Il Worker avvia la fase di distruzione dell'ambiente locale
9: Il Worker termina la sua esecuzione finalizzando il sistema IPC.

Quando tutti i Worker hanno ricevuto la segnalazione di EoS ed hanno terminato la loro esecuzione:

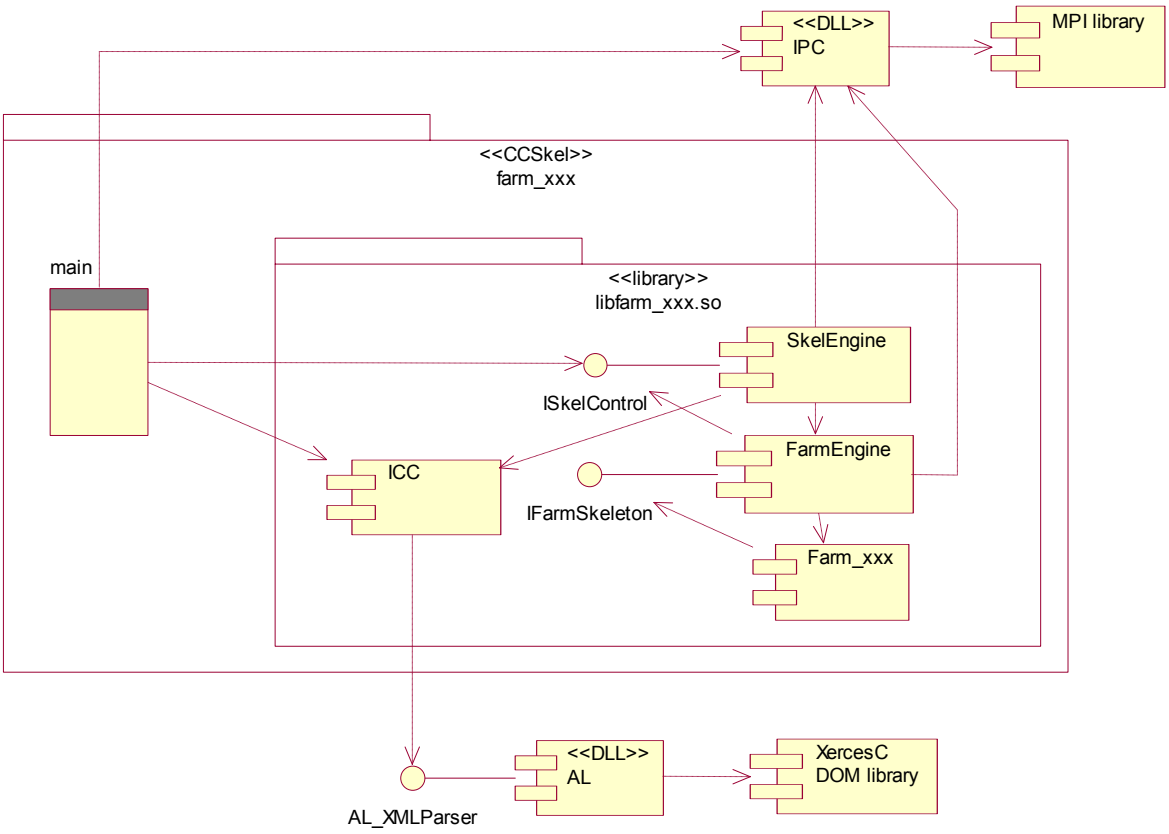
10: L'Emitter avvia la fase di distruzione della struttura dati di Output.
11: L'Emitter avvia la fase di distruzione della struttura dati di Input.

12: L’Emitter avvia la fase di distruzione dell’ambiente locale
13: L’Emitter termina la sua esecuzione finalizzando il sistema IPC.

Il componente termina la sua esecuzione.

5. Architettura di deployment di un componente CCskel

Il componente parallelo che implementa lo skeleton Farm ha la seguente architettura:



Il componente parallelo CCSkel (**farm_xx**) è un eseguibile attivabile da riga di comando. Tramite argomenti *argc,argv* è possibile *configurare* il componente in termini di *grafo di esecuzione* e *parametri di attivazione*.

- Il grafo di esecuzione specifica l'insieme dei nodi su cui avviare il componente, in termini di nodi, collegamenti e relativi indici di performance e velocità, in linguaggio XML.
- I parametri di attivazione corrispondono alla rimanente parte degli argomenti e sono passati in blocco all'emitter in quanto determinano l'input del componente.

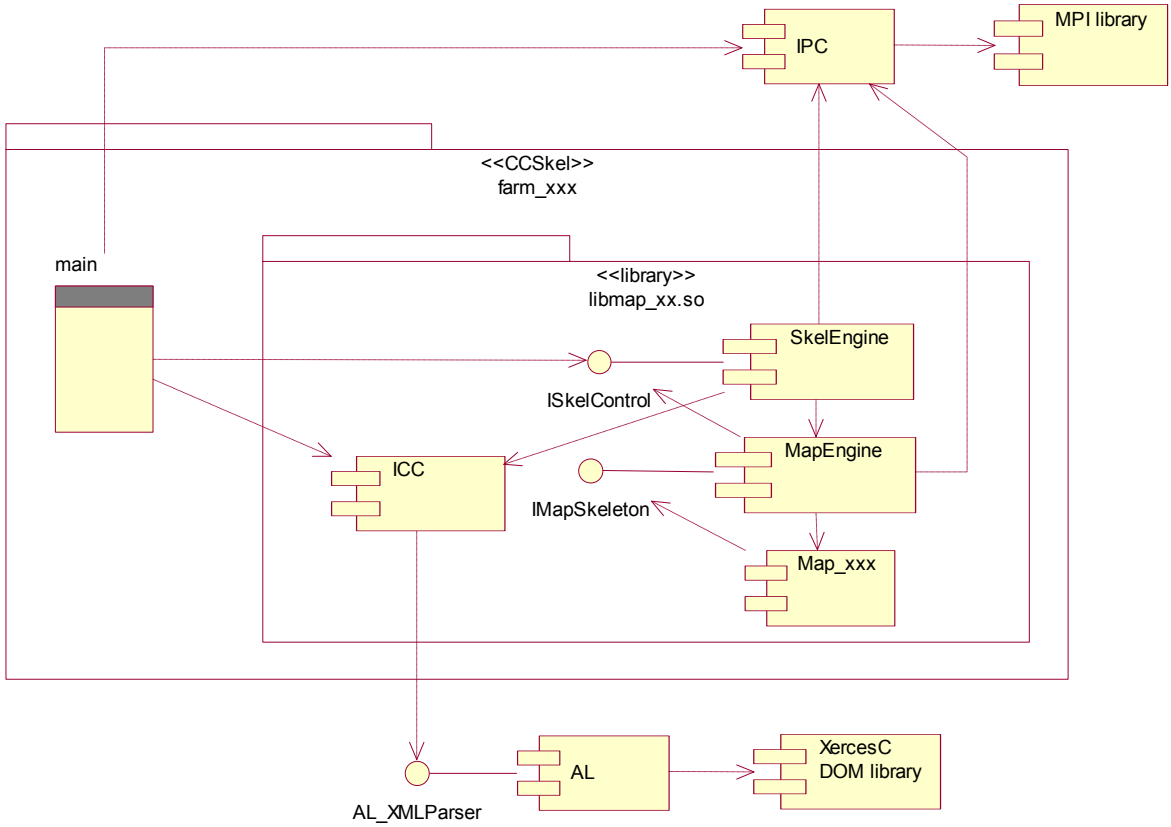
Il componente usa la libreria parallela (**libfarmxx.so**) attraverso l'interfaccia **SkelEngine** che permette l'avvio di uno skeleton generico (Farm/Map). Nella libreria parallela, il package **FarmEngine** ne realizza un' implementazione di tipo Farm e definisce la API per l'esecuzione di ciascuna delle fasi previste nei processi Emitter e Worker. Il package **Farm_xxx** ne realizza un'implementazione secondo l'applicazione specifica.

Ogni implementazione di tale libreria è, di fatto, il componente parallelo in quanto contiene la parte di codice dipendente dall'applicazione specifica. Il Package SkelEngine permette la corretta esecuzione delle varie fasi implementate nel FarmEngine secondo le due possibilità Emitter o Worker. Il **main** del CCSkel effettua l'avvio di una delle due possibilità di esecuzione in un particolare nodo del grafo.

Il package **ICC** fornisce il supporto per l'interpretazione dell'XML contenente la configurazione del componente. Essa è implementata tramite Apache-*XercesC DOM*.
La libreria **IPC** fornisce il supporto per l'*attivazione* del componente e lo *scambio di messaggi* a run-time. Essa è implementata tramite il sistema **MPI.** In particolare le primitive fornite sono:

- **init**, per l'attivazione del componente
- **send**, per l'invio di un messaggio
- **recv**, per la ricezione di un messaggio
- **barrier**, per la sincronizzazione tra processi
- **finalize**, per la finalizzazione del componente.

Il componente parallelo che implementa lo skeleton Map ha la seguente architettura:



Il componente parallelo CCSkel (**map_xx**) usa la libreria parallela (**libmapxx.so**) attraverso l'interfaccia **SkelEngine** che permette l'avvio di uno skeleton generico (Farm/Map). Nella libreria parallela, il package **MapEngine** ne realizza un' implementazione di tipo Map e definisce la API per l'esecuzione di ciascuna delle fasi previste nei processi Emitter e Worker. Il package **Map_xxx** ne realizza un'implementazione secondo l'applicazione specifica.
Ogni implementazione di tale libreria è, di fatto, il componente parallelo in quanto contiene la parte di codice dipendente dall'applicazione specifica. Il Package SkelEngine permette la corretta esecuzione delle varie fasi implementate nel MapEngine secondo le due possibilità Emitter o Worker. Il main del CCSkel effettua l'avvio di una delle due possibilità di esecuzione in un particolare nodo del grafo.

Bibliografia

[1] M. Vanneschi: Proposta di Progetto CNR, sottoprogetto Grid Computing: Tecnologie Abilitanti e Applicazioni per eScience, Fondo Speciale 1999

[2] M. Vanneschi, "PQE2000 : HPC tools for industrial applications". IEEE Concurrency, IEEE Computer Society, October-December 1998, pp. 68 – 73.

[3] M. Vanneschi: The programming model of ASSIST, an environment for parallel and distributed portable applications. Parallel Computing 28(12): 1709-1732 (2002)

[4] M. Vanneschi, "Heterogeneous HPC environments", invited paper, *4th Int. Euro-Par Conference*, Southampton, Sept. 1998, in D. Pritchard and J. Reeve (Eds.), *Lecture Notes in Computer Science*, vol. 1470, pp. 21-34.

[5] Foster, I. Kesselman, C. Nick, J. and Tuecke S. : The Anatomy of the Grid: Enabling Scalable Virtual Organizations. Intern. Journal of High performance Computing Applications., 15(3), 200-222.

[6] .Foster, I. Kesselman, C. Nick, J. and Tuecke S.: The Physiology of the Grid:An Open Grid Services Architecture for Distributed Systems Integration. Global Grid Forum, Open Grid Service Infrastrucutre WG http://www.globus.org.. Intern. Journal of High performance Computing Applications., 15(3), 200-222

[7] Rob Armstrom, Dennis Gannon, Katarezyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinsk. "*Toward a common component architecture for high-performance scientific computing*". In Conference on High Performance Distributed Computing, 1999

[8] The Common Component Architecture Technical Specification – Version 0.5. http://cca-forum.org/bindings/old-0.5/.

[9] F.Berman, G.C. Fox, A.J.G.Hey: *Grid Computing: Making the Global Infrastructure a Reality*. Wiley 2003

[10] Zizhong Chen, Jack Dongarra, Piotr Luszczek, and Kenneth Roche"Self Adapting Software for Numerical Linear Algebra and LAPACK for Clusters www.cs.utk.edu/~luszczek/ articles/lfc-parcomp.pdf.

[11] Machì, S. Lombardo "A conceptual model for grid-adaptivity of HPC applications and its logical implementation with components technology" Accepted for presentation at ICSSA04/AGCPA Krakow, Poland June 2004. TR ICAR-PA-13-03 - Dec 2003

[12] M. Cole, Algorithmic Skeletons: Structured Management of Parallel Computation, The MIT Press, Cambridge, Massachusetts,1989.

[13] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, R. L. While, Q. Wu, *Parallel Programming using Skeleton Functions*, In A. Bode, M. Reeve and G. Wolf, editors, *Proc. of PARLE'93*, volume 694 of *LNCS*, pages 146-160, Springer-Verlag, 1993.

[14] J. Darlington, Y. Guo, H. W. To, J. Yang, *Parallel skeletons for structured composition*, In Proc. of the 5th ACM/SIGPLAN Symposium on Principles and Practice of Paral1el Programming, Santa Barbara, California, July 1995, *SIGPLAN Notices* 30(8),19-28.G. Sardisco,

[15] A. Machi': "Development of Parallel Paradigms Templates For Semi-automatic Digital Film Restoration Algorithms" Proc. PARCO 2001, Naples Sept. 2001.