Éléments d'architecture client-serveur

Michel Simatic

module CSC4508/M2

Avril 2018

Plan du document

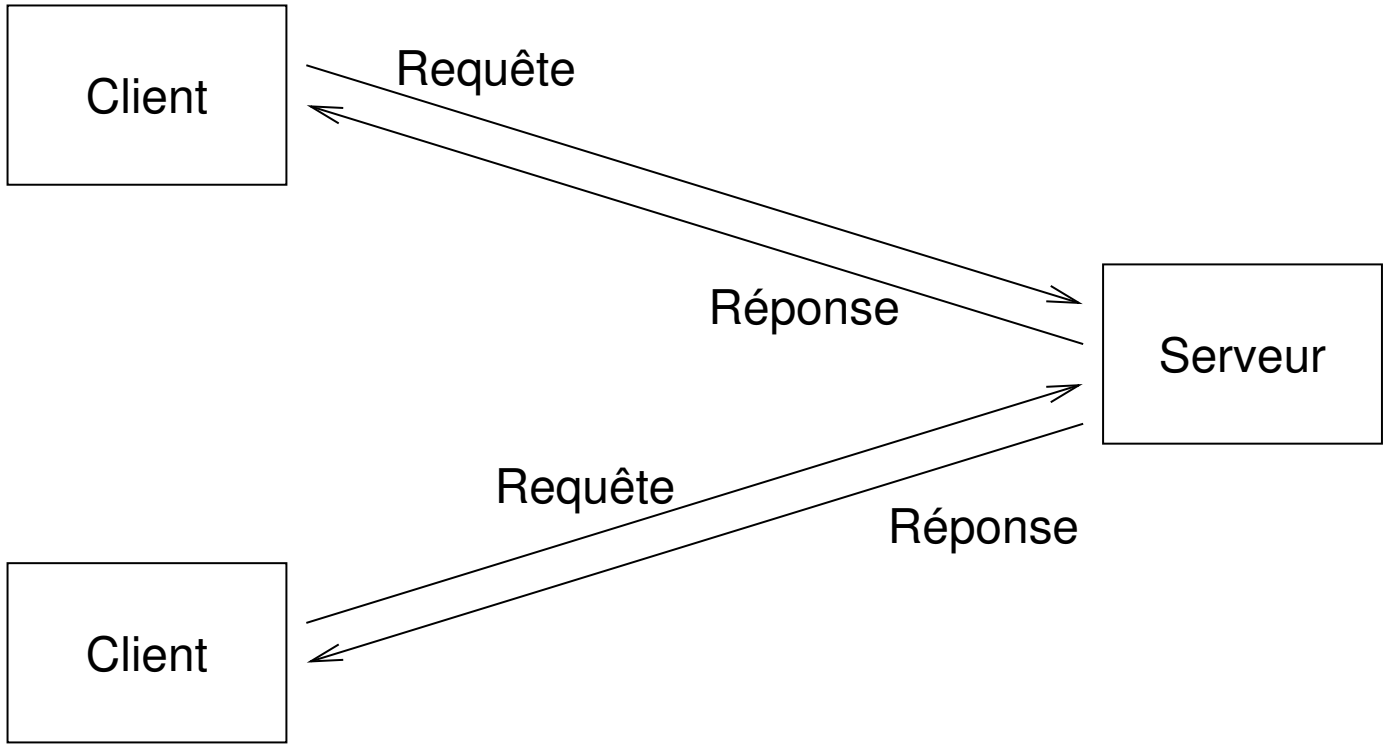
1 Introduction 3
2 Serveur mono-tâche gérant un client à la fois 7
3 Serveur avec autant de tâches que de clients 10
4 Serveur avec N tâches gérant tous les clients 15
5 Serveur mono-tâche gérant tous les clients à la fois 18
6 Conclusion 23

1 Introduction

1.1 Définition d'une architecture client/serveur 4
1.2 Objectif de cette présentation 5
1.3 À propos des communications 6

1.1 Définition d'une architecture client/serveur

- Une application fonctionne selon une architecture client-serveur quand : les machines « clientes » contactent une machine « serveur » afin que ce serveur leur fournisse un service (via l'exécution d'un programme)
- Les clients envoient des requêtes
- Le serveur envoie des réponses



1.2 Objectif de cette présentation

- Étudier différentes architectures logicielles pour traiter le flux de requêtes arrivant au niveau du serveur

  4 types d'architecture :

  - Serveur mono-tâche gérant un client à la fois
  - Serveur avec autant de tâches que de clients
  - Serveur avec N tâches gérant tous les clients
  - Serveur mono-tâche gérant tous les clients à la fois
- Analyser les qualités et les défauts de chacune de ces architectures
- Étudier comment combiner les différentes briques du système d'exploitation pour disposer de l'architecture la plus propice à répondre aux besoins de l'application

1.3 À propos des communications

- Tout processus (client ou serveur) peut recevoir des messages via un (ou plusieurs) « point(s) d'accès » qui lui est (sont) propre(s)
- Pour qu'un autre processus lui envoie un message via un « point d'accès », il faut, au préalable, que cet autre processus se « connecte » (au sens protocolaire du terme) (exemple : TCP, tube nommé) ou non (exemple : UDP, Système de signalisation CCITT N° 7)
- Par abus de langage, on dit aussi qu'un client se « connecte » au serveur lorsqu'il a une suite d'échanges requête/réponse avec le serveur (cette « connexion client » nécessitant une connexion protocolaire ou non).

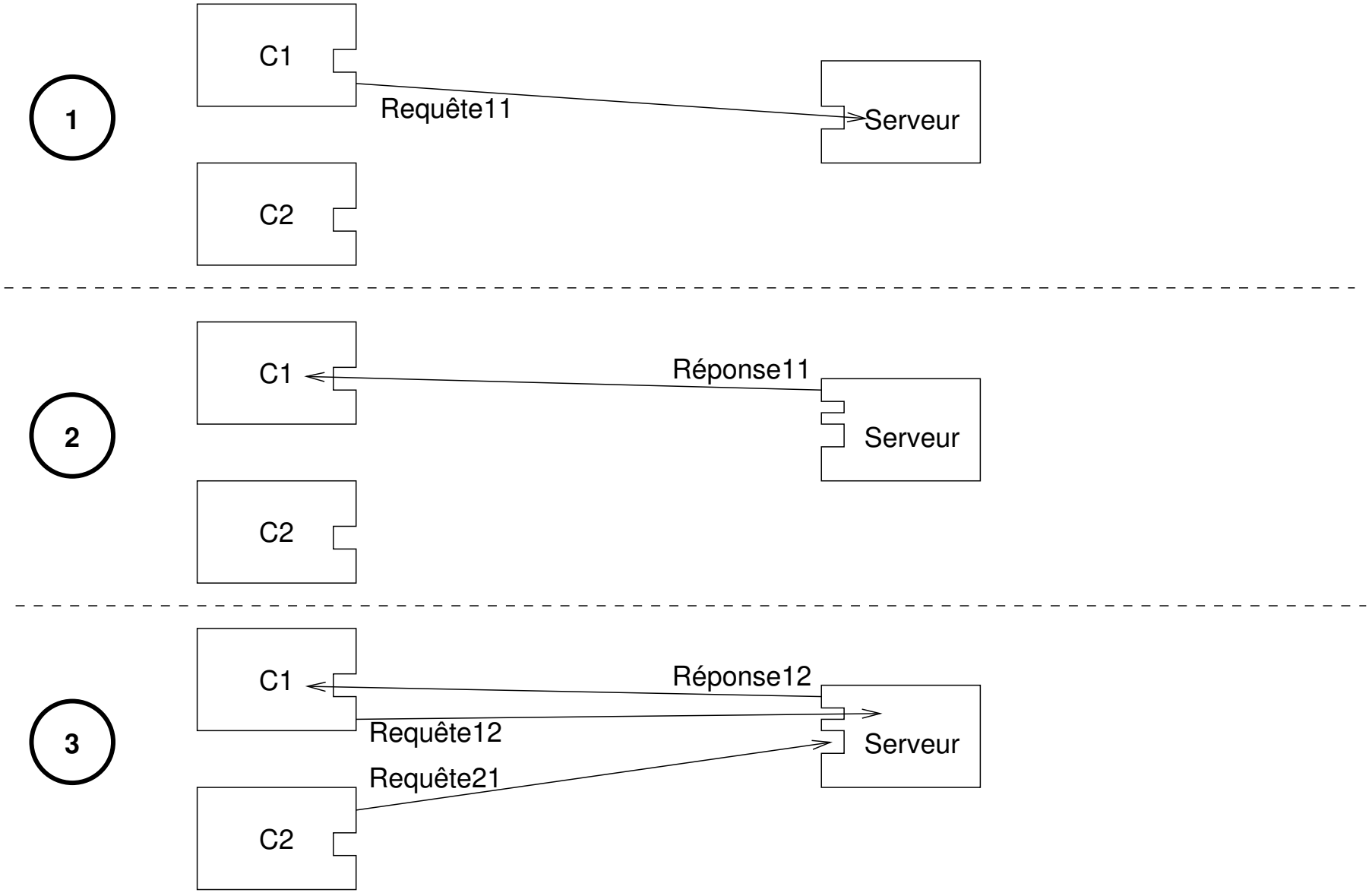
  Dans la suite de cette présentation, le terme de « connexion » désignera toujours une « connexion client »
- L'aspect communication inter-machine est hors du contexte de ce cours. Donc, dans les TPs, on utilisera des « files de messages » en guise de « points d'accès »

2 Serveur mono-tâche gérant un client à la fois

2.1 Principe 8
2.2 Analyse 9

2.1 Principe

- La tâche serveur traite les connexions client de bout en bout



2.2 Analyse

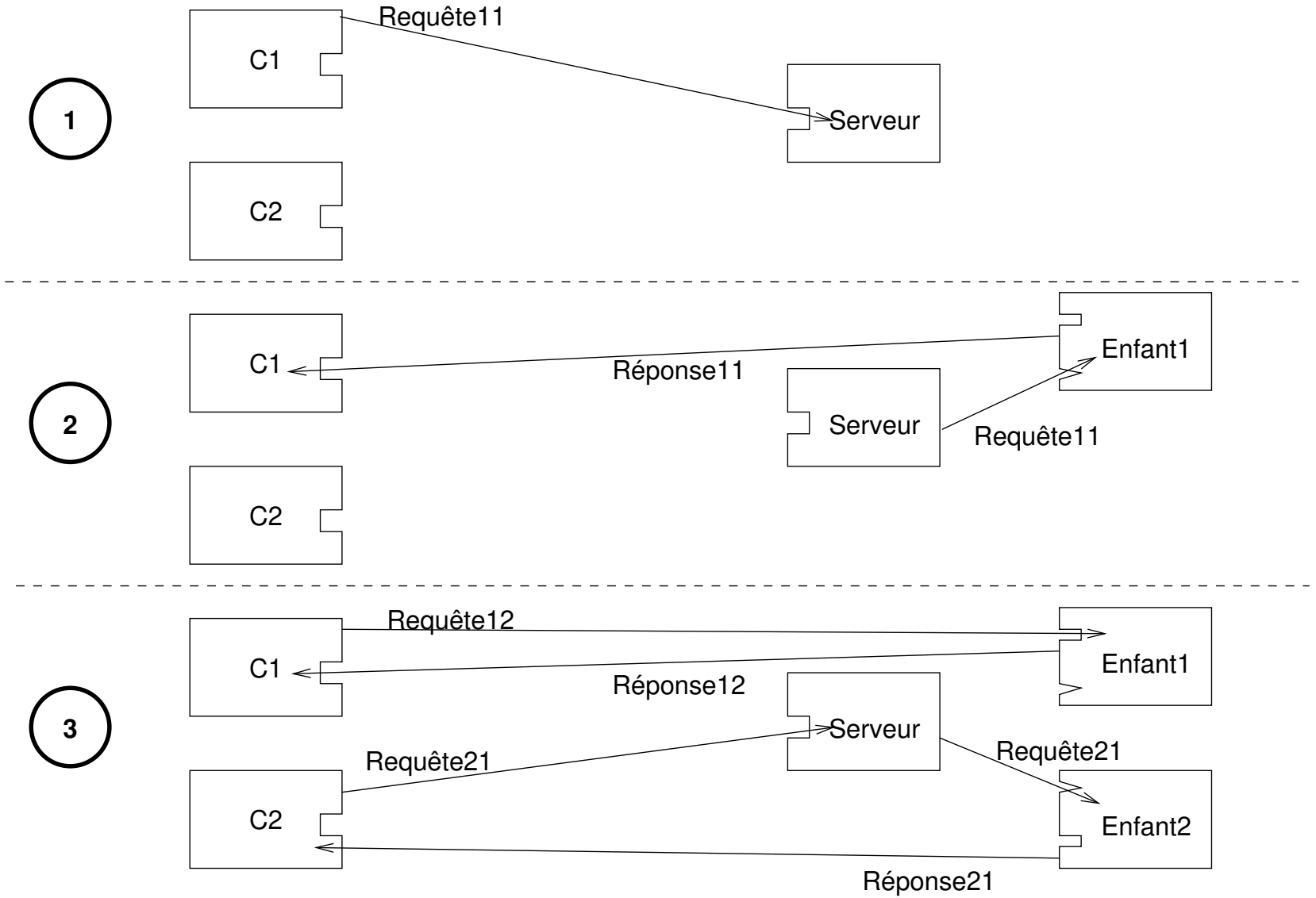
- Avantages
  - Architecture simple
  - Bien adaptée au cas de traitements courts
  - Machine serveur peu chargée
  - Pas de risque de surcharge
- Inconvénients
  - Architecture inadaptée au cas de traitements longs, que cette longueur soit dûe à :
    - Temps de traitement long au niveau du serveur
    - Nombreux échanges requête/réponse entre client et serveur avant que le client ne se déconnecte

3 Serveur avec autant de tâches que de clients

3.1 Principe 11
3.2 Variante 12
3.3 Analyse 13
3.4 Réduction du temps de connexion des clients 14

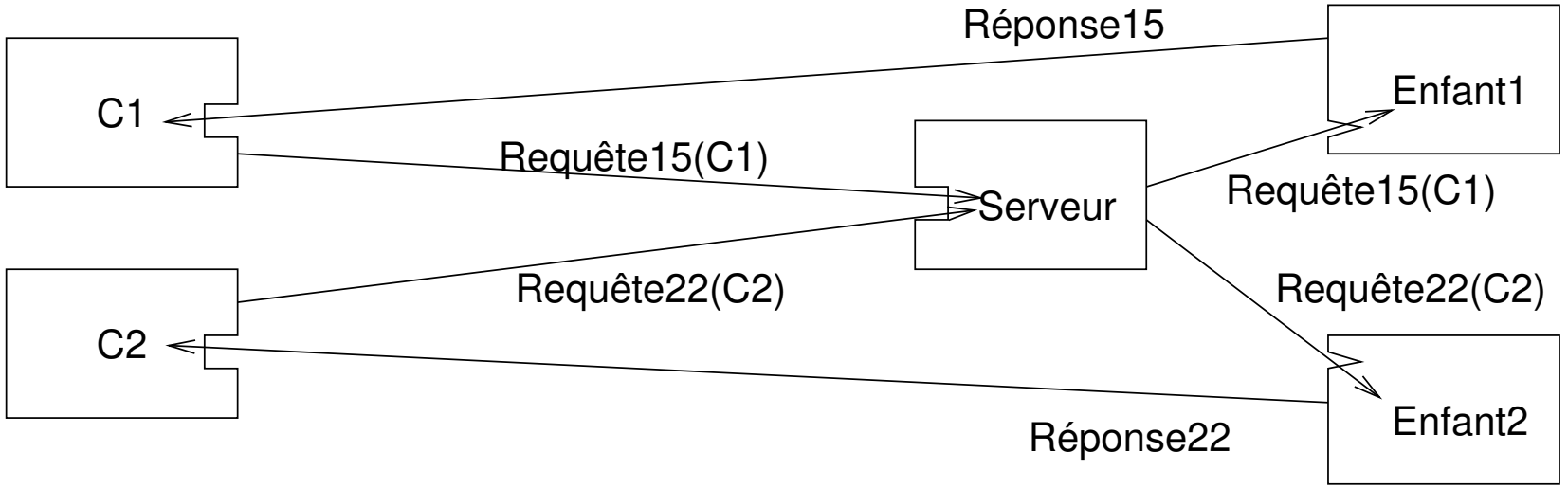
3.1 Principe

- La tâche serveur délègue les connexions client à des enfants



3.2 Variante

- Dans le cas où le nombre de « points d'accès » est limité au niveau de la machine du serveur (exemple : SS7), la tâche serveur sert de distributeur à ses enfants
- On parle alors de « multiplexage » des « connexions client » au niveau du serveur



3.3 Analyse

- Avantages
  - Adapté au cas de traitements longs
- Inconvénients
  - Le serveur doit être hébergé par une machine puissante
  - Risque de surcharge à moins d'utiliser :
    - des mécanismes de synchronisation de tâches à base de « cohorte »
    - un *pool* d'enfants disponibles
  - Sensibilité au nombre de clients : la charge du serveur augmente proportionnellement au nombre de clients

    Solutions possibles :
    - Augmenter la puissance du serveur (par exemple, grappes de PC)
    - Réduire le temps de connexion des clients

3.4 Réduction du temps de connexion des clients

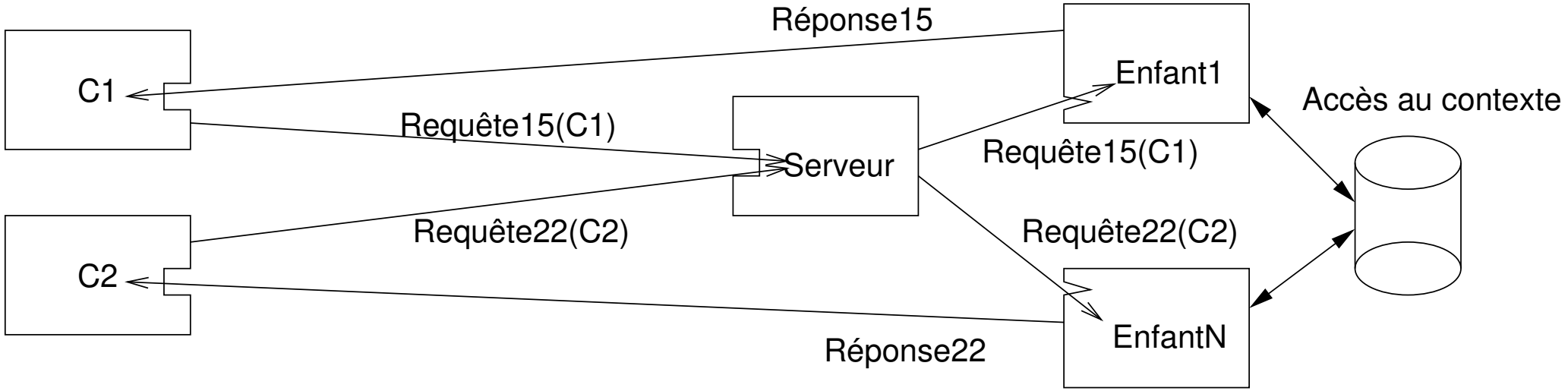
- Principe:
  - L'enfant n'est connecté avec le client que pendant le temps de traitement d'une requête
  - Si le client a une requête ultérieure *faisant partie du même contexte*, il transmet le contexte avec la requête ultérieure
- Exemple : Google
- Limite : Solution inapplicable au cas d'applications où le client ne peut héberger le contexte :
  - Soit pour des raisons de capacité (mémoire, traitement. . .)
  - Soit pour des raisons de sécurité

4 Serveur avec N tâches gérant tous les clients

4.1 Principe 16
4.2 Analyse 17

4.1 Principe

- Le serveur héberge dans une « base de données »[a] le contexte de chacun de ses clients potentiels
- Au démarrage, le serveur démarre N enfants
- Lors de la réception d'une requête, le serveur confie à l'un de ses enfants la tâche de gérer la requête compte tenu du contexte client
- L'enfant fait éventuellement évoluer le contexte



a. Cette base de données peut être une zone de mémoire partagée

4.2 Analyse

- Avantages
  - Adapté au cas de connexions longues avec de longs temps d'inactivité
  - Cas, par exemple, des applications télécoms[a]
- Inconvénients
  - Complexité
  - Risques de surcharge

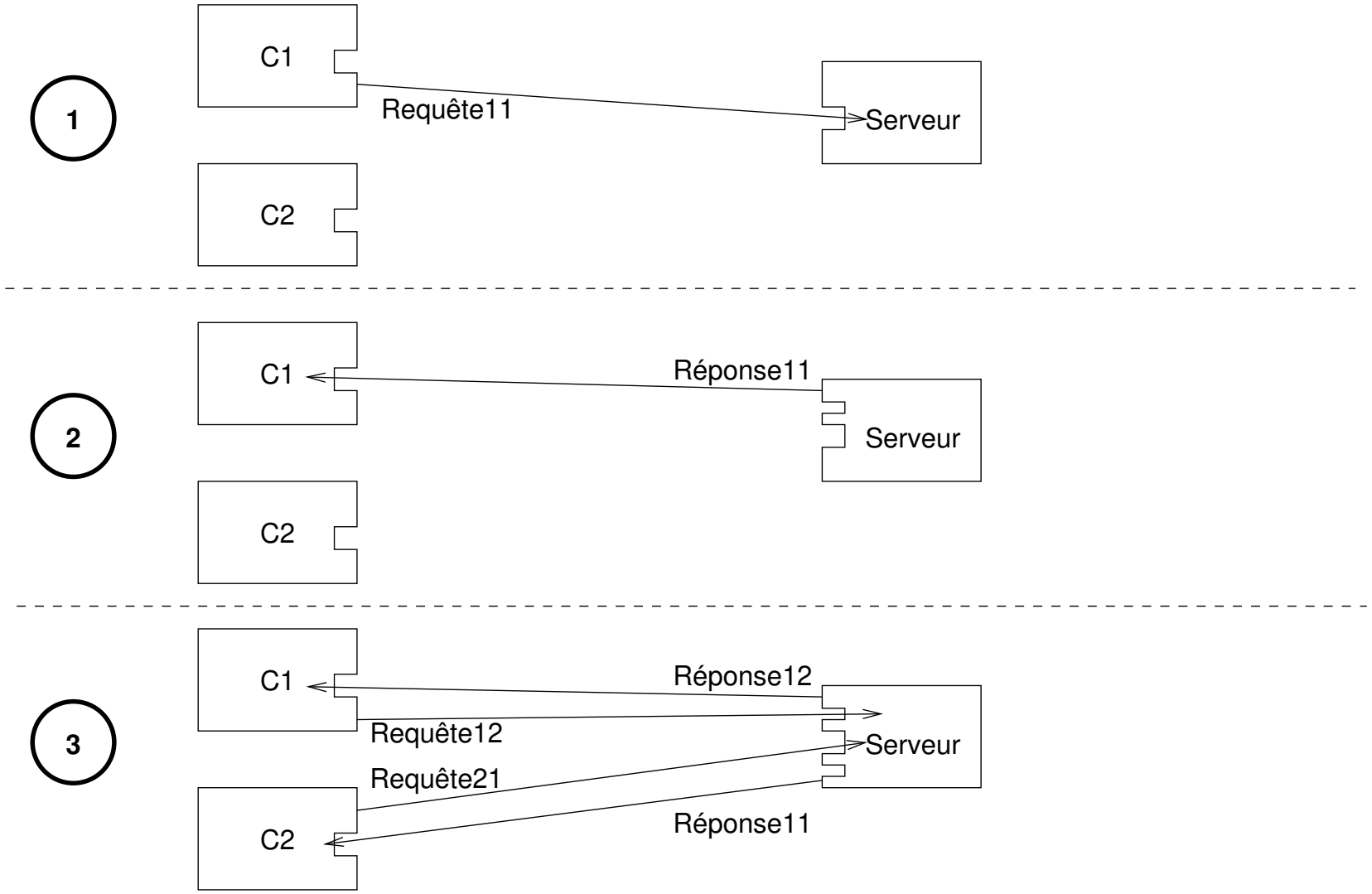
a. Le gros du traitement d'appel se fait à l'établissement et au raccrochage (le temps de conversation qui est le plus long en durée ne consomme pratiquement rien en ressources serveur)

5 Serveur mono-tâche gérant tous les clients à la fois

5.1 Principe 19
5.2 Aperçu de la programmation événementielle 20
5.3 Analyse 22

5.1 Principe

- La tâche serveur travaille avec un modèle de programmation événementielle.



5.2 Aperçu de la programmation événementielle

- Principe
  - On ne déclenche une opération de lecture sur un descripteur que quand on est sûr qu'il y a effectivement quelque chose à lire sur ce descripteur.
  - Pour savoir si c'est le cas, on s'appuie sur des primitives système comme `select`, `poll`, `epoll`...

- Des intergiciels (*middleware*) facilitent le développement d'applications basées sur ce modèle. Ils garantissent la portabilité entre systèmes différents tout en utilisant les primitives système les plus performantes.

  Exemple : `libevent` (`http://monkey.org/~provos/libevent/`)
  - `event_set` permet de spécifier le descripteur à surveiller et la fonction de rappel (*callback*) à invoquer si un événement arrive sur le descripteur.
  - `event_add` permet de donner cette spécification à la tâche principale de `libevent`.

- `event_dispatch` démarre la tâche principale de `libevent`. Si un événement arrive sur un descripteur, cette tâche appelle la fonction *callback* associée.

5.3 Analyse

- Avantages
  - Adapté au cas de connexions longues avec de longs temps d'inactivité
- Inconvénients
  - Complexité
  - Risques de surcharge
  - Le traitement d'une requête ne doit pas comprendre des appels-systèmes lents (à moins que ce ne sois des entrées-sorties) : le serveur se comporterait alors comme un serveur mono-processus gérant un client à la fois

6 Conclusion

- Ce cours a présenté 4 architectures logicielles de serveur. Elles mettent en œuvre tous les mécanismes système étudiés au cours de cette U.V.

- Quelle est la meilleure architecture ?

  La réponse à cette question dépend de critères dont l'importance varie selon le projet considéré:

  - Nombre maximum de clients simultanés à gérer,
  - Durée de traitement d'une requête au niveau du serveur,
  - Expérience des développeurs,
  - Importance de la dimension *Green Computing*,
  - ...

Bibliographie du chapitre